AD-A245 549

ATION PAGE

Form Approved
OPM No. 0704-0188

Public re... needed, ... Headqua... Managem... ...r response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data ...n estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington ...on Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of ...

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 16 Nov 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE
Rational, Rational Environment, D_12_24_0, R1000 Series (Host & Target), 901116W1.11084

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-427-1290

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

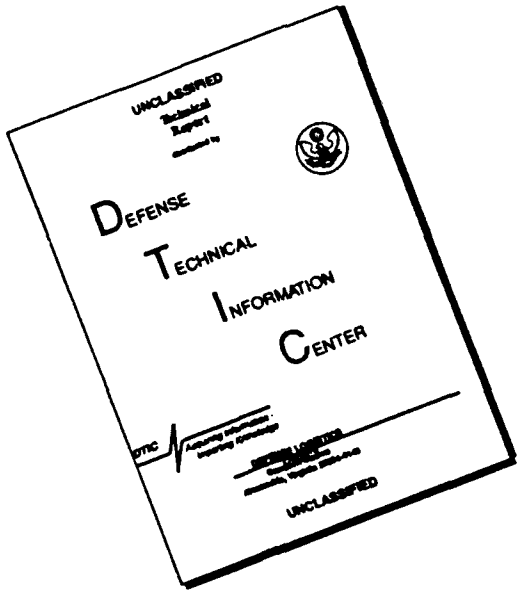
12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Rational, Rational Environment,Wright-Patterson AFB, D_12_24_0, R1000 Series (Host & Target), ACVC 1.11.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1990.

Compiler Name and Version: Rational Environment, Version D_12_24_0

Host Computer System: R1000 Series 300,
Rational Environment, Version D_12_24_0

Target Computer System: R1000 Series 300,
Rational Environment, Version D_12_24_0

Customer Agreement Number: 90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11084 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accesion For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

AVF Control Number: AVF-VSR-427-1290
19 November 1991
90-07-20-RAT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901116W1.11084
Rational
Rational Environment, D_12_24_0
R1000 Series 300 => R1000 Series 300

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 16 November 1990.

Compiler Name and Version: Rational Environment, Version D_12_24_0

Host Computer System: R1000 Series 300,
Rational Environment, Version D_12_24_0

Target Computer System: R1000 Series 300,
Rational Environment, Version D_12_24_0

Customer Agreement Number: 90-07-20-RAT

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901116W1.11084 is awarded to Rational. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

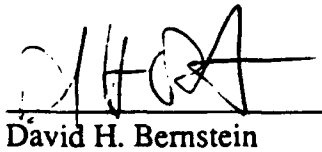
Customer:	Rational
Ada Validation Facility:	ASD/SCEL, Wright-Patterson AFB OH 45433-6503
ACVC Version:	1.11

Ada Implementation

Compiler Name:	Rational Environment, D_12_24_0
Host Architecture:	R1000 Series 300
Host Operating System:	Rational Environment Version D_12_24_0
Target Architecture:	R1000 Series 300
Target Operating System:	Rational Environment Version D_12_24_0

Customer's Declaration

I, the undersigned, representing Rational, declare that Rational has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Rational is the owner of the above implementation and the certificates shall be awarded in the name of the owners corporate name.



Date: 9/27/90

David H. Bernstein
Vice President, Products Group

Rational
3320 Scott Blvd.
Santa Clara, CA 95054

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 USE OF THIS VALIDATION SUMMARY REPORT 1-1
1.2 REFERENCES 1-2
1.3 ACVC TEST CLASSES 1-2
1.4 DEFINITION OF TERMS 1-3

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS 2-1
2.3 TEST MODIFICATIONS 2-5

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-1
3.3 TEST EXECUTION 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August, 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of Identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35702B, C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

D64005G uses 17 levels of recursive procedure calls nesting which exceeds the capacity of the compiler.

B86001Y checks for a predefined fixed-point type other than DURATION.

C92005B includes a conversion of the 'STORAGE_SIZE value for a task to type INTEGER; for this implementation, that value exceeds INTEGER'LAST and the conversion raises CONSTRAINT_ERROR, which terminates the test. (See section 2.3.)

LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F check for pragma INLINE for procedures and functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

BD2A85A and BD2A85B expect that sizes of 2 and 3, respectively, will be rejected as too small for the size of an access type when given in a length clause; this implementation treats access values as offsets and so accepts the length clauses. (See section 2.3.)

The following 45 tests contain address clauses. This implementation does not support address clauses. (See section 2.3.)

CD5003A..I (9 tests)	CD5011A	CD5011C
CD5011E	CD5011G	CD5011I
CD5011K	CD5011M	CD5011Q
CD5011S	CD5012A..B (2 tests)	CD5012E..F (2 tests)
CD5012I	CD5012M	CD5013A
CD5013C	CD5013E	CD5013G
CD5013I	CD5013K	CD5013M
CD5013O	CD5014A	CD5014C
CD5014E	CD5014G	CD5014I
CD5014K	CD5014M	CD5014O
CD5014T	CD5014Y	CD5014X..Z (3 tests)

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO

CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	----------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2102K and CE2401B check operations on files with an element type of the access class; this implementation does not permit I/O of access values, and so raises USE_ERROR on the attempt to create a file.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2401B checks READ, WRITE, SET_INDEX, INDEX, SIZE, and END_OF_FILE for direct files with ELEMENT_TYPES boolean, access, and enumerated.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 111 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B22003B	B22004A	B22004B	B22004C	B23002A
B23004A	B23004B	B24001A	B24001B	B24001C	B24005A
B24005B	B24007A	B24009A	B24204B	B24204C	B24204D
B25002B	B26001A	B26002A	B26005A	B28003A	B28003C
B29001A	B2A003B	B2A003C	B2A003D	B2A007A	B32103A
B33201B	B33202B	B33203B	B33301A	B33301B	B35101A
B36002A	B37106A	B37205A	B37307B	B38003A	B38003B
B38009A	B38009B	B41201A	B44001A	B44004A	B44004B
B44004C	B44004D	B44004E	B45205A	B48002A	B48002D
B53003A	B55A01A	B56001A	B63001A	B63001B	B64001B
B64006A	B67001A	B67001B	B67001C	B67001D	B67001H
B71001A	B71001G	B71001M	B74003A	B74307B	B83E01C
B83E01D	B83E01E	B91001F	B91001H	B91003E	B95001D
B95003A	B95004A	B95006A	B95007B	B95079A	BA1001B
BB3005A	BC1109A	BC1109B	BC1109C	BC1109D	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B	BC3013A
BE2210A	BE2413A	B51001A			

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

BD2A85A and BD2A85B were graded inapplicable by Evaluation Modification as directed by the AVO. These tests each contain a size length clause for an access type that is expected to be rejected by the compiler because of the small value that is specified for the size. However, this implementation treats access values as offsets into the collection (see Appendix C, section 4.6), and thus accepts these clauses.

C45232A was passed by Test Modification as directed by the AVO. This test contains the expression "INTEGER'LAST > SMALL_INT'BASE'LAST" at lines 131 and 169; the test does not anticipate that if the condition is false, the implicit conversion of the right operand to type INTEGER may raise an exception. This implementation selects type LONG_INTEGER for the base type of SMALL_INT, and so raises an exception. The test was modified by inserting ' FALSE THEN --' immediately after 'IF' in both lines, which avoids the exeption and accurately reflects the actual condition; this modified version was passed.

A99007A was graded passed by Test Modification as directed by the AVO. This test assigns a task's 'STORAGE_SIZE value to an INTEGER object; for this implementation that assignment raises an exception, because the value

is greater than INTEGER'LAST. The test was modified by re-defining type "INTEGER" at line 13 with "TYPE INTEGER IS RANSGE 0..MAX_INT;"; the modified test was passed.

C92005B was graded inapplicable by Evaluation Modification as directed by the AVO. This implementation's 'STORAGE_SIZE value for tasks is 2**32, unless another value has been specified by a length clause. In this test no length clause is present, and the conversion of attribute's value to type INTEGER raises CONSTRAINT_ERROR; there is no handler for this exception in the test, and so execution terminates.

CD1C04E was graded passed by Evaluation Modification as directed by the AVO. This test checks that a record representation clause for a derived type dtermines the values of the 'POSITION, 'FIRST_BIT, and 'LAST_BIT attributes of components of objects of the type, when the parent type has been given a different representation. For this implementation, SYSTEM.STORAGE_UNIT = 1, and thus 'FIRST_BIT always returns 0 and the check for inequality fails. The AVO ruled that this was acceptable behavior; the test was graded passed bacause all other checks were applicable and passed.

CC1220A was graded passed by Evaluation Modification as directed by the AVO. This test evaluates the address of the same generic formal object of mode in at lines 35 and 66; it expects that address to be the same. The issues of what the address of a constant is, and whether it must remain constant over the life of the object are unclear; the AVO ruled that the implementation's behavior is acceptable pending resolution of AI-00203 by the ARG. The test was graded as passed even though the test reported "FAILED", given that the only call to Report.Failed occurred for the controversial check at line 66, which output the message "IMPROPER VALUE FOR 'ADDRESS".

CD2A83A, CD2A84A, CD2A84E, CD2A84I, CD2B11A, and CD2B11B were graded passed by Test Modification as directed by the AVO. These tests check the use of access types whose type size and collection size have been specified with length clauses. In order to accommodate this implementation's unusual interpretation of these values (see the discussion above re BD2A85A and BD2A85B), the specified values were modified as follows and the modified tests were passed:

for CD2A84A, CD2A84E, and CD2A84I, changed were:

constant BASIC_SIZE's value, from 8 to 10, 11, and 10, respectively;
constant COLL_SIZE's value, by inserting the multiplier ' * 8'
(which essentially compensates for SYSTEM.STORAGE_UNIT being 1 vs. 8)

for CD2A83A, constant COLL_SIZE's value was also changed as above; and

for CD2B11A and CD2B11B, constant BASIC_SIZE's value was changed by inserting ' * 8' (in these two tests, BASIC_SIZE is the collection size)

The following 45 tests were graded inapplicable by Evaluation Modification as directed by the AVO. These tests check for support of address clauses. This implementation "uses a memory-protection scheme that prohibits

arbitrary address calculations including the use of an address literal. ... Consequently, address clauses are not supported." [Appendix C, section 6.1]

CD5003A..I (9 tests)	CD5011A	CD5011C
CD5011E	CD5011G	CD5011I
CD5011K	CD5011M	CD5011Q
CD5011S	CD5012A..B (2 tests)	CD5012E..F (2 tests)
CD5012I	CD5012M	CD5013A
CD5013C	CD5013E	CD5013G
CD5013I	CD5013K	CD5013M
CD5013O	CD5014A	CD5014C
CD5014E	CD5014G	CD5014I
CD5014K	CD5014M	CD5014O
CD5014T	CD5014V	CD5014X..Z (3 tests)

BD4007A, BD4009A, and CD4051C were graded passed by Test Modification as directed by the AVO. These tests use record representation clauses that place a field at a 0 offset from the start of a record with discriminants; but this implementation reserves a prefix of at least 1 bit at the head of any record with discriminants, and so rejects the representation. The tests were modified by inserting '1' to change the offsets to 10 vice 0; the modified tests were passed.

> For each of the affected component clauses (lines 47..49 and 56..57; 26, 27 and 29; and 29 of the tests, respectively) insert '1' immediately before the '0', changing the offset value to "10".

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

For a point of contact for sales information about this Ada implementation system, see:

David H. Bernstein
3320 Scott Blvd.
Santa Clara CA 95054

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3739	
b) Total Number of Withdrawn Tests	81	
c) Processed Inapplicable Tests	149	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	350	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 350 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option | Switch	Effect
Create_Subprogram_Specs = False	When a library unit subprogram body is added to the program library, do not automatically create a corresponding subprogram specification.
Retain_Deltal_Compatibility = False	Do not generate code compatible with Deltal version of the Rational Enviroment.
Ignore_Interface_Pragmas = True	Causes pragma Interface to be ignored.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN—also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	254
$ACC_SIZE	24
$ALIGNMENT	1
$COUNT_LAST	1000000000
$DEFAULT_MEM_SIZE	268435456
$DEFAULT_STOR_UNIT	1
$DEFAULT_SYS_NAME	R1000
$DELTA_DOC	1.08420217248550443400745280086994171142578125OE-19
$ENTRY_ADDRESS	ADDRESS1
$ENTRY_ADDRESS1	ADDRESS2
$ENTRY_ADDRESS2	ADDRESS3
$FIELD_LAST	2147483647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	5.0E09
$GREATER_THAN_DURATION_BASE_LAST	3.0E14
$GREATER_THAN_FLOAT_BASE_LAST	2.0E308
$GREATER_THAN_FLOAT_SAFE_LARGE	2#1111111111111.1111111111111111111111111111111111111111111.0#E971

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E308

$HIGH_PRIORITY          5

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BAD_CHARACTERS&<>=

$ILLEGAL_EXTERNAL_FILE_NAME2
                        CONTAINS_WILDCARDS@

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -2147483647

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     NO_LANGUAGE

$LESS_THAN_DURATION     -5.0E09

$LESS_THAN_DURATION_BASE_FIRST
                        -5.0E14

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        INSTRUCTION'(ACTION,IDLE);

$MACHINE_CODE_TYPE      INSTRUCTION

$MANTISSA_DOC           63

$MAX_DIGITS             18

$MAX_INT                9223372036854775807

$MAX_INT_PLUS_1         9223372036854775808

$MIN_INT                -9223372036854775808
```

MACRO PARAMETERS

$NAME	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST	R1000
$NAME_SPECIFICATION1	!VALIDATION.ACVC_1_11.R1000.C_LOGS.CHAPTER_E.X2120A
$NAME_SPECIFICATION2	!VALIDATION.ACVC_1_11.R1000.C_LOGS.CHAPTER_E.X2120B
$NAME_SPECIFICATION3	!VALIDATION.ACVC_1_11.R1000.C_LOGS.CHAPTER_E.X3119A
$NEG_BASED_INT	16#FFFFFFFFFFFFFFFF#
$NEW_MEM_SIZE	268435456
$NEW_STOR_UNIT	1
$NEW_SYS_NAME	R1000
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	INSTRUCTION
$TASK_SIZE	64
$TASK_STORAGE_SIZE	1024
$TICK	200.0E-9
$VARIABLE_ADDRESS	ADDRESS4
$VARIABLE_ADDRESS1	ADDRESS5
$VARIABLE_ADDRESS2	ADDRESS6
$YOUR_PRAGMA	ENABLE_DEALLOCATION

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

```
PROCESSOR                SWITCH                          TYPE          VALUE

 R1000_Cg . Asm_Listing                          : Boolean          := False
-- Enables the creation of R1000 assembly code list files.  Each file is
-- stored as an attribute of the associated Ada unit.

 R1000_Cg . Check_Compatibility                  : Boolean          := True
-- Check that subsystem load views are compatible with the corresponding
-- spec views when loading programs.

Directory . Create_Internal_Links                : Boolean          := True
-- Controls whether internal links are created automatically when the
-- visible parts of library units are created.  Internal links for library
-- units are created in the set of links for the nearest enclosing world.
-- The default is True.  The full switch name is
-- Directory.Create_Internal_Links.  (For further information on links, see
-- the Key Concepts section of the Library Management (LM) Reference
-- Manual.) .

* Directory . Create_Subprogram_Specs            : Boolean          := False
-- Controls whether specifications for library-unit subprograms are created
-- automatically.  The contents of these specifications are created the
-- first time the body is successfully installed.  The "with" clause for
-- the specification is derived from the "with" clauses in the body. Only
-- those "with" clauses required to promote the specification are included.
--  The default is True.  The full switch name is
-- Directory.Create_Subprogram_Specs.

 R1000_Cg . Delta1_Code_View_Compatibility : Boolean          := False
-- Causes Compilation.Load and Cmvc.Make_Code_View to create additional
-- objects (i.e., pre-D_11_3_0 code archives) that will be needed if and
-- only if the resulting Loaded Main Program or Code View is to be copied
-- or restored onto another R1000 running a pre-D_11_3_0 environment
```

```
— release; for this switch to have any effect, the
— Retain_Delta1_Compatibility switch should also be set.

 R1000_Cg . Elab_Order_Listing              : Boolean          := False
— No help available for this switch.

 R1000_Cg . Enable_Deallocation             : Boolean          := False
— Implicitly apply the Enable_Deallocation pragma to all access types so
— that storage can be reclaimed using Unchecked_Deallocation.

 R1000_Cg . Full_Debugging                  : Boolean          := False
— Causes the code generator to emit code so that breakpoints can be set at
— all Ada source constructs, (e.g., null statements).

 R1000_Cg . Package_Integration             : Boolean          := False,
— Causes the code generator to integrate all packages for which
— integration is possible (i.e., assume a pragma Integrate was associated
— with all integrable packages). When a package is integrated, the code
— is generated as if all of its declarations were declared directly with
— its parent package (no architectural module is created for an integrated
— package). Package integration is not allowed for packages containing
— tasks, packages containing an initialization block, library unit
— packages or package subunits whose bodies are separate.

 R1000_Cg . Page_Limit                      : Integer          := 8000
— Maximum number of pages for segments in jobs which contain the unit;
— default is 8000 pages.

Directory . Require_Internal_Links          : Boolean          := True
— Controls whether failure to create internal links (as controlled by the
— Directory.Create_Internal_Links switch) generates an error. The default
— (True) is to treat the failure to generate links as an error and to
— discontinue the operation. If the Directory.Create_Internal_Links
— switch is set to False, this switch has no effect. The full switch name
— is Directory.Require_Internal_Links.

 R1000_Cg . Retain_Delta1_Compatibility     : Boolean          := True
— Causes the R1000 code generator to emit code that is spec_view/load_view
— compatible with code generated under previous releases of the
— environment (releases before D_11_3_0). A unit that was coded with the
— switch set one way may freely reference a second unit that was coded
— with the switch set the other way, but corresponding units in a spec
— view and a load view of a given subsystem must be coded with the switch
— set the same way. A unit that was coded under a pre-D_11_3_0 release of
— the environment is considered to have been coded with the switch set to
— True. Enabling the switch also causes the code generator to reject some
— valid Ada constructs (e.g., constraining a incomplete discriminated
— type) and to generate incorrect code for some others (e.g., uses of
— exceptions declared within generic units).

 R1000_Cg . Seg_Listing                     : Boolean          := False
— Enables the creation of R1000 object code list files. Each file is
— stored as an attribute of the associated Ada unit.
```

```
R1000_Cg . Terminal_Echo                    : Boolean                    := False
-- If Asm_Listing is set, then the assembly code listing will be sent to
-- the terminal as the code is being generated.
```

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
    ..........
    type Integer is range -2147483647 .. 2147483647;
    type Long_Integer is range -9223372036854775808 .. 9223372036854775807;

    type Float is
       digits 15
          range -1.79769313486228199394645344000309705734252?296875E308 ..
                   1.79769313486228199394645344000309705734252?296875E308;

    type Duration is
       delta 3.0517578125E-5
          range -4294967296.0 .. 4294967295.999969482421875;
    ..........
end STANDARD;
```

Appendix F for the R1000 Target

Appendix F describes the implementation-dependent features of the Ada language, as it is implemented for the Rational Environment™, the Rational architecture, and the R1000® target. If you are using a Cross-Development Facility (CDF) to compile programs for a non-R1000 target, refer to the Appendix F that is provided with the documentation for that CDF.

Appendix F is a required part of the *Reference Manual for the Ada Programming Language* (LRM). The following sections are required by the LRM:

1. "Implementation-Dependent Pragmas" describes the form, allowed places, and effect of every implementation-dependent pragma.
2. "Implementation-Defined Attributes" describes the name and type of every implementation-dependent attribute.
3. "Predefined Language Environment" presents the specifications of package System and package Standard.
4. "Support for Representation Clauses" lists all of the restrictions on representation clauses.
5. "Implementation-Generated Names" describes the conventions used for any implementation-generated name denoting implementation-dependent components of records.
6. "Address Clauses" describes the interpretation of expressions that appear in address clauses, including those for interrupts.
7. "Unchecked Programming" describes any restrictions on unchecked conversions and deallocations.
8. "Input/Output Packages" describes any implementation-dependent characteristics of the input/output packages.

The following sections contain additional information about the R1000 compiler:

9. "Capacity Restrictions" describes implementation-dependent limits on various aspects of program size.
10. "Attributes of Numeric Types" lists the values returned by attributes that apply to integer types, floating-point types, and the fixed-point type Duration.
11. "Compilation in the Rational Environment" briefly describes some of the concepts that underlie the Rational Environment compilation system, provides a summary of the separate compilation rules for Ada units in the Environment, and lists the Environment library switches that can affect compilation.

1. IMPLEMENTATION-DEPENDENT PRAGMAS

The Environment accepts the pragmas defined in Annex B of the LRM, as well as a number of additional pragmas to be used in application software development. The first of the following two sections lists clarifications and restrictions for the pragmas defined in Annex B. The second of these two sections describes the Environment-defined pragmas.

1.1. Pragmas Defined in Annex B

For each of the pragmas defined in Annex B of the LRM, this section (Table 1) describes the extent to which it is supported for the R1000 target. Support for these pragmas may differ for other targets. Information about the pragmas that are supported for each non-R1000 target is given in the CDF manual for that target.

Table 1 Pragmas Defined in Annex B

Pragma	Effect
Controlled	Always implicitly in effect for the R1000 target because the implementation does not support automatic garbage collection.
Elaborate	Fully supported.
Inline	Has no effect for the R1000 target.
Interface	Has no effect for the R1000 target because the Environment does not support the execution of other languages on the Rational architecture. The specific behavior of this pragma depends on the value of the Semantics.Ignore_Interface_Pragmas library switch at compilation time. When this switch is True, the R1000 compiler ignores the pragma completely. When this switch is False, the R1000 compiler builds an implicit body that raises the Program_Error exception whenever the subprogram is executed.
List	Has no effect for the R1000 target because compiler listings are not generated.
Memory_Size	This pragma has no effect for the R1000 target because memory size cannot be changed.
Optimize	Has no effect for the R1000 target.
Pack	Always implicitly in effect for the R1000 target because all records and arrays are stored packed in the minimum number of bits that they require.
Page	Has no effect on the compiler, but is interpreted by the print spooler to cause a new page. The pragma marks the last line on the page. The next line is printed on the next page.
Priority	Priorities can be specified only inside a task or a library main program. If multiple priorities are specified, only the first priority specified is used. The default priority is 1. The range is 0 .. 5.
Shared	Has no effect for the R1000 target.
Storage_Unit	Has no effect for the R1000 target because the only legal storage-unit value is 1.
Suppress	Suppresses only Elaboration_Check. This pragma has no effect if any other check is specified.
System_Name	Has no effect for the R1000 target because the only legal system name is R1000.

1.2. Environment-Defined Pragmas

For each of the pragmas defined by the Environment, this section describes the extent to which it is supported for the R1000 target. Support for these pragmas may differ for other targets. Information about the pragmas that are supported for each non-R1000 target is given in the CDF manual for that target.

Table 2 summarizes the Environment-defined pragmas described in this section.

Table 2 Environment-Defined Pragmas

Pragma	Effect
Case_Method	Specifies one of three search methods to be used to find the selected alternative in a case statement or variant in a variant record.
Disable_Deallocation	Disables the deallocation of storage for the specified access type.
Enable_Deallocation	Enables the deallocation of storage for the specified access type.
End_Of_Unit	Specifies by its position the boundary between adjacent compilation units that are stored in a text file.
Loaded_Main	Indicates that a unit is a loaded main program. This pragma is automatically generated by the Environment and never entered by users.
Main	Instructs the Environment to create a coded main program when the unit containing it is promoted to the coded state.
Open_Private_Part	Causes a given package specification to have an open private part, causing the private part to be compiled whenever the package is compiled.
Page_Limit	Specifies the minimum value for the page limit of a job executing the unit containing the pragma.
Private_Eyes_Only	Causes the context clauses following the pragma to be ignored when the unit specification containing the pragma is compiled.

1.2.1. Case_Method

Pragma Case_Method specifies one of three search methods to be used to find the selected alternative in a case statement or variant in a variant record. This pragma must appear before the first alternative of the case statement or before the first variant of the variant record. If this pragma is omitted, the Environment will choose an appropriate method or combination of methods.

This pragma has the form:

```
pragma Case_Method (search_method);
```

where ***search_method*** is one of the following:

- **Linear_Search**, which causes the choices to be compared against the selector sequentially in the order in which they occur in the Ada source.

- **Binary_Search**, which causes the choices to be sorted at compile time and a binary search to be used at run time to select the appropriate alternative. If all choices are equally probable and there are more than five choices, this method is faster than linear search.
- **Jump_Table**, which causes the selector to be used to index into a jump table. This is the fastest method but may require large amounts of space if the range of choices is large.

For example:

```
case X is
    pragma Case_Method (Binary_Search);
    when 1       => ...
    when 10      => ...
    when 101     => ...
    when others => ...
end case;
```

1.2.2. Disable_Deallocation

Pragma Disable_Deallocation specifies the name of an access type for which deallocation is disabled. Disabling deallocation for an access type prevents the Unchecked_Deallocation procedure from reclaiming storage for that type. This pragma is used when deallocation has been enabled for the entire library (that is, the library switch R1000_Cg.Enable_Deallocation is set to True) and you want to disable deallocation for a particular access type. (See also Section 7, later in this appendix.) This pragma has the form:

```
pragma Disable_Deallocation (access_type_name);
```

and must occur immediately within the same declarative part as the access type to which it applies. Within this declarative part, pragma Disable_Deallocation can be placed anywhere following the declaration of the specified access type.

1.2.3. Enable_Deallocation

Pragma Enable_Deallocation specifies the name of an access type for which deallocation is enabled. This pragma enables the Unchecked_Deallocation procedure to reclaim storage for the specified access type. This pragma is used when deallocation has been disabled for the entire library (that is, the library switch R1000_Cg.Enable_Deallocation is set to False) and you want to enable deallocation for a particular access type. (See also Section 7, later in this appendix.) This pragma has the form:

```
pragma Enable_Deallocation (access_type_name);
```

and must occur immediately within the same declarative part as the access type to which it applies. Within this declarative part, pragma Enable_Deallocation may be placed anywhere following the declaration of the specified access type. Note that this pragma also can be used on a generic formal to indicate that it should be deallocatable.

1.2.4. End_Of_Unit

Pragma End_Of_Unit specifies by its position the boundary between adjacent compilation units that are stored in a text file. This pragma has the form:

```
pragma End_Of_Unit;
```

For example, assume you are transferring compilation units from a text-based computer system to the R1000 and that one of the transferred text files contains multiple compilation units. Assume further that comments and pragmas occur between two adjacent compilation units in this file. Then, before you use Compilation.Parse to parse this file, you can insert pragma End_Of_Unit between the adjacent units to clarify which of the comments and pragmas belong at the end of the preceding unit and which belong to the beginning of the following unit. After the units have been parsed, pragma End_Of_Unit is automatically removed.

If you omit this pragma, the Environment parses interunit pragmas and comments by dividing them at the first full-line comment (a comment that stands alone on a line and is not the continuation of a right-trailing comment above it). All pragmas and comments up to (but not including) such a comment are put at the end of the preceding compilation unit, and the remaining pragmas and comments are put at the beginning of the following compilation unit.

1.2.5. Loaded_Main

Pragma Loaded_Main appears only when generated by the Environment and is never entered by users. The Environment generates this pragma to specify that a unit is a . A loaded main program is an executable program that is not dependent on its source code; that is, a loaded main program will not be obsolesced if the source code from which it was created is modified. Furthermore, a loaded main program can be moved between R1000s without having to move and recompile its source code. This pragma appears in loaded main programs in the following form:

```
pragma Loaded_Main;
```

You can create a loaded main program from a coded main program using the Compilation.Load procedure. This procedure creates a separate, self-contained copy of the main program's code segments, similar to an executable module on other computer systems. The Compilation.Load procedure automatically inserts the Loaded_Main pragma in place of the Main pragma in the newly created loaded main program.

Because its code segments are independent from its source code, a loaded main program is unaffected by demoting, and even changing, the source code. Thus, the consistency between a loaded main program and its source code may be lost. Furthermore, loaded main programs, like code views, can be debugged using the Rational debugger only if the same version of the original source code still exists in the same location on the same R1000 and is still in the coded state.

1.2.6. Main

Pragma Main instructs the Environment to create a when the main subprogram containing the pragma is promoted to the coded state. The resulting coded main program has *prelinked* object code for the main subprogram and all of the units in its closure. In other words, elaboration information is generated at the time the main subprogram is coded and then retained for use whenever the coded main program is executed. When pragma Main is omitted from a program, elaboration information is generated (and subsequently discarded) each time the program is executed. Thus, coded main programs execute faster than nonmain programs, so that using pragma Main is recommended for large, frequently executed programs.

Pragma Main must be placed immediately after the end of the specification or body of a main subprogram. A main subprogram must be a library-unit subprogram that has no subunits and cannot be a generic or a generic instantiation. A main subprogram's parameters (and result, if any) must be of types defined in package Standard, in package System, or in basic Environment-supplied packages. Before a main subprogram can be promoted to the coded state, all compilation units in the closure of the main subprogram also must be in the coded state.

This pragma has the following form, where the three optional parameters are described below:

```
pragma Main (Target           => target_name,
             Activity         => "activity_file_name",
             Elaboration_Order => "file_name");
```

- **Target**: Specifies the name of the target for which the pragma is intended. This is useful during cross-development, when you may want to compile a unit as a coded main program for one target but not for others. This parameter controls the applicability of the pragma in the following way: if the specified name matches the target key of the enclosing library, the pragma is used; otherwise, the pragma is ignored by the current target compiler. Leaving this parameter unspecified causes the pragma to be used wherever the unit is compiled (that is, the parameter value defaults to the target referenced by the current target key). Thus, if you want a unit to be compiled as a coded main program for the R1000 target, you must either specify this parameter with the value R1000 or leave the Target parameter unspecified (depending on whether additional targets are to be considered as well).

  If a unit is to be compiled as a coded main program for each of several targets, the unit can contain one pragma Main for each target. When the unit is compiled for one of these targets, the compiler filters out all but the pragma Main whose Target parameter matches the target key; if none of the pragma Mains specifies the current target key, the unit is not compiled as a main program. Thus, you can compile the unit for different targets without having to edit the source code to add or remove the pragma.

- **Activity**: Specifies the name of the activity to be used when generating elaboration code for the main program. If a simple name is specified, it is resolved relative to the library containing the main program. If the Activity parameter is omitted, the default activity for the current session is used. Note that an activity is consulted only when units in the closure of the main subprogram are defined in subsystems.

- **Elaboration_Order**: Specifies the name of a text file that contains a list of unit names in the order in which they are to be elaborated. The specified elaboration order is subject to normal Ada elaboration-ordering rules; the purpose of this parameter is to allow you to specify a

particular order where a choice can be made. If a simple name is specified, it is resolved relative to the library containing the main program.

The contents of the specified file must have the same format as indirect files. Typically, this means that each line contains a valid naming expression. The specified file need not be a complete list of all units in the program; rather, this file can list just the units whose order is to be specified. Note that the required indirect file format is used in the elaboration-order listing files that are produced whenever the R1000_Cg.Elab_Order_Listing switch is set to True in libraries containing main subprograms. Consequently, you can modify an elaboration-order listing file to reflect the desired elaboration order and then specify it through this parameter.

To understand pragma Main in more detail, it is necessary to understand what takes place before the Environment actually executes a program. In particular, the Environment:

1. Checks the compilation units in the closure of the program to verify that they exist, are complete, and are in the coded state. Note that for programs developed in subsystems, the current (or specified) activity is used to determine the actual units in the program's closure.
2. Computes a valid elaboration order for the closure of the program, based on the rules specified in LRM 10.5. (If applicable, the file specified by the Elaboration_Order parameter of pragma Main is also taken into account.)
3. Generates elaboration code that, when executed, causes all of the units in the program's closure to be elaborated in the order computed in step 2.

These steps are referred to as *prelinking*. For programs that do not contain pragma Main, prelinking occurs when coded units are promoted for execution. In this case, the elaboration information is discarded after it is executed. For programs that contain pragma Main, prelinking occurs earlier—specifically, when the main subprogram is promoted to the coded state. The computed elaboration order and elaboration code are retained for use each time the program is executed. Note that after the prelinking has taken place, main programs written in subsystems execute without consulting an activity.

The retained elaboration information is discarded and the main program is demoted to installed if any of the units in the program are demoted below the coded state. In other words, a coded main program maintains a dependency on its source code. To remove this dependency, you can use the Compilation.Load command to create a loaded main program from a coded main program; see Section 1.2.5, above.

Bear in mind that units in the closure of a coded main program are actually elaborated when the main program is called. If a command or program calls other coded main programs, and if the closures of these coded main programs share any units, then separate copies of the shared units are elaborated.

1.2.7. Open_Private_Part

Pragma Open_Private_Part causes a given package specification to have an open private part. This pragma has the form:

```
pragma Open_Private_Part;
```

and is effective only in package specifications defined in spec views. Furthermore, this pragma must occur before the first private type in a given package specification. The effect of pragma Open_Private_Part is not inherited by nested packages; this pragma must be included in each package that is to have an open private part.

By default, packages in spec views have closed private parts, which support the conceptual separation of private parts from visible package declarations. When packages with closed private parts are compiled in a spec view, the private parts of these packages are ignored by the compiler. Consequently, the private parts of exported packages can be changed in a load view without rendering that load view incompatible with the compiled spec view (and without requiring the corresponding changes to be made in the spec view, which would necessitate the recompilation of client views).

Pragma Open_Private_Part overrides the default treatment of private parts in spec-view packages, causing the private parts to be compiled whenever the packages containing them are compiled. This allows the compiler to take advantage of the specific type information in the private parts, resulting in more efficient code. However, clients of packages with open private parts must be recompiled whenever any of the private parts are changed. Furthermore, changes to private parts must be made in both the spec and load views to preserve compatibility between these views. Finally, pragma Private_Eyes_Only has no effect when private parts are open.

Pragma Open_Private_Part also may appear in package specifications that are defined in load views or in libraries outside subsystems. However, because private parts are always open in such views and libraries, the pragma is ignored. For more information, see the "Key Concepts" section in the Project Management (PM) book of the *Rational Environment Reference Manual*.

1.2.8. Page_Limit

Pragma Page_Limit specifies the minimum value for the page limit of a job executing the unit containing the pragma. The page limit is the number of virtual memory pages (containing 1024 bytes each) that can be created by the job that elaborates the unit in which the pragma appears.

This pragma has the form:

```
pragma Page_Limit (positive_integer);
```

and is allowed after the end of a unit specification or body.

Note that the value of this pragma is ignored if it is less than either of the values given by the library switch R1000_Cg.Page_Limit and the session switch Default_Job_Page_Limit. In other words, the compiler uses the maximum of the values given by the pragma and these two switches. For a more detailed description, see the reference entries for the System_Utilities.Get-_Page_Counts and System_Utilities.Set_Page_Limit procedures in the System Management Utilities (SMU) book of the *Rational Environment Reference Manual*.

1.2.9. Private_Eyes_Only

Pragma Private_Eyes_Only causes the context clauses following the pragma to be ignored when the unit specification containing the pragma is compiled. Such context clauses can be ignored only if they are required only by closed private parts in a unit specification in a subsystem spec view. For more information, see the "Key Concepts" section in the Project Management (PM) book of the *Rational Environment Reference Manual*.

This pragma has the form:

```
pragma Private_Eyes_Only;
```

and has no effect in any of the following situations:

- When private parts are open, as in units containing pragma Open_Private_Part
- In generic units, for which the private parts are always open
- In units located outside subsystems

2. IMPLEMENTATION-DEFINED ATTRIBUTES

This section describes the implementation-defined attributes for the R1000 target.

2.1. P'Compiler_Version

For a prefix P that denotes an object, type, exception, package, or subprogram, this attribute yields the version of the compiler that was used to generate code for the unit containing the definition of P. The value returned by this attribute is of type string—for example, "11.4.0".

2.2. P'Declaration_Number

For a prefix P that denotes an object, type, exception, package, or subprogram, this attribute yields the *declaration number* that is associated with the declaration of P. Within a subsystem, every declaration in a given library-unit specification is identified by a unique declaration number. The value returned by this attribute is a Long_Integer.

Taken together, the values of P'Subsystem_Number, P'Unit_Number, and P'Declaration_Number provide a unique way of identifying a given declaration P in a given library-unit specification in a given subsystem. See also T'Type_Key.

2.3. P'Subsystem_Number

For a prefix P that denotes an object, type, exception, package, or subprogram, this attribute yields the *subsystem identification number* of the subsystem containing the unit in which P is declared. Every primary subsystem on an R1000 has a unique subsystem identification number; in fact, a given subsystem identification number is unique across all R1000s. Note that secondary subsystems always have the same subsystem identification number as the primary subsystem with which they are associated, even if they reside on different machines. The value returned by this attribute is a Long_Integer.

Taken together, the values of P'Subsystem_Number, P'Unit_Number, and P'Declaration_Number provide a unique way of identifying a given declaration P in a given library-unit specification in a given subsystem. See also T'Type_Key.

2.4. P'Target_Key

For a prefix P that denotes an object, type, exception, package, or subprogram, this attribute yields the portion of the target key that indicates the compiler that was used to generate code for the unit containing the definition of P. The value returned by this attribute is of type string—for example, "R1000".

2.5. T'Type_Key

For a prefix T denoting a type name, this attribute yields a string representation of the triple consisting of the subsystem identification number, the unit number, and the declaration number for T—for example, "612.14.19". Thus, the string returned by this attribute uniquely identifies type T. This attribute typically is used when passing messages of a given type over a network to ensure that the reader and writer agree on the type to use when interpreting the message.

Similarly, this attribute is used when storing values of the type in a file to ensure that the reader and writer agree on the type to use when interpreting data in the file.

See also the P'Subsystem_Number, P'Unit_Number, and P'Declaration_Number attributes, which return the integer representations of the individual components of the T'Type_Key value.

2.6. P'Unit_Number

For a prefix P that denotes an object, type, exception, package, or subprogram, this attribute yields the ***unit number*** of the unit in which P is declared. Within a given subsystem, every library unit is identified by a unique unit number. The value returned by this attribute is a Long_Integer.

Taken together, the values of P'Subsystem_Number, P'Unit_Number, and P'Declaration_Number provide a unique way of identifying a given declaration P in a given library-unit specification in a given subsystem. See also T'Type_Key.

3. PREDEFINED LANGUAGE ENVIRONMENT

This section presents the implementation-dependent portions of package System and package Standard. Note that these packages are presented in an elided form, generally omitting declarations that are not implementation-dependent or that are reserved for internal use only. Additional information about packages Standard and System is in the Programming Tools (PT) book of the *Rational Environment Reference Manual*.

3.1. Package System

Package System defines various implementation-dependent types, objects, and subprograms.

```
package System is

    type Address is private;
    Null_Address : constant Address;

    type Name    is (R1000);
    System_Name  : constant Name := R1000;

    Bit          : constant :=    1;
    Storage_Unit : constant :=    1 * Bit;
    Word_Size    : constant :=  128 * Bit;
    Byte_Size    : constant :=    8 * Bit;
    Megabyte     : constant := (2 ** 20) * Byte_Size;
    Memory_Size  : constant := 32 * Megabyte;

    -- System-dependent named numbers

    Min_Int      : constant := Long_Integer'Pos (Long_Integer'First);
    Max_Int      : constant := Long_Integer'Pos (Long_Integer'Last);
```

```
    Max_Digits    : constant := 15;
    Max_Mantissa  : constant := 63;
    Fine_Delta    : constant := 1.0 / (2.0 ** 63);
    Tick          : constant := 200.0E-9;

    subtype Priority is Integer range 0 .. 5;

    -- Bytes are basic units of transmission/reception to/from I/O devices
    type Byte is new Natural range 0 .. 255;
    type Byte_String is array (Natural range <>) of Byte;

    -- Exceptions raised by Unchecked_Conversion or Unchecked_Conversions;
    -- may also result from spec/load view incompatibilities or
    -- from compiler/microcode errors

    Type_Error        : exception;
    Capability_Error  : exception;
    Assertion_Error   : exception;

    -- Exceptions resulting from spec/load view incompatibilities or
    -- from compiler/microcode errors

    Frame_Establish_Error    : exception;
    Heap_Pointer_Copy_Error  : exception;
    Illegal_Instruction      : exception;
    Illegal_Frame_Exit       : exception;
    Illegal_Heap_Access      : exception;
    Illegal_Reference        : exception;
    Invalid_Package_Value    : exception;
    Machine_Restriction      : exception;
    Microcode_Assist_Error   : exception;
    Nonexistent_Page_Error   : exception;
    Nonexistent_Space_Error  : exception;
    Operand_Class_Error      : exception;
    Record_Field_Error       : exception;
    Select_Use_Error         : exception;
    Unsupported_Feature      : exception;
    Utility_Error            : exception;
    Visibility_Error         : exception;
    Write_To_Read_Only_Page  : exception;

end System;
```

3.2. Package Standard

Package Standard defines all of the predefined identifiers in the language.

```
package Standard is

    type Boolean is (False, True);
```

```
    type Integer is range -2147483647 .. 2147483647;
    subtype Natural is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;

    type Long_Integer is range -9223372036854775808 .. 9223372036854775807;

    type Float is
       digits 15
          range -1.79769313486231E+308 .. 1.79769313486231E+308

    type Duration is
       delta 3.0517578125E-5
          range -4.29496729600000E+09 .. 4.29496729599997E+09

    type Character is ...;
    for Character'Size use 8;
    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    package Ascii is ... end Ascii;

    Constraint_Error : exception;
    Numeric_Error    : exception;
    Storage_Error    : exception;
    Tasking_Error    : exception;
    Program_Error    : exception;

end Standard;
```

4. SUPPORT FOR REPRESENTATION CLAUSES

This section describes the Environment's support for representation clauses for the R1000 target. In particular, this section describes the default representation of each type (that is, the representation that is used in the absence of representation clauses) as well as the values that can be used in the representation clauses for these types. The various types are listed below, along with the kinds of representation clauses that apply to them:

- Integer types: size length clauses
- Enumeration types: size length clauses and enumeration representation clauses
- Floating-point types: size length clauses
- Fixed-point types: size length clauses and small length clauses
- Access types: size length clauses and storage-size length clauses
- Task types: size length clauses and storage-size length clauses
- Array types: size length clauses
- Record types: size length clauses and record representation clauses

Note that, in addition to representation clauses, the LRM permits another mechanism for controlling the representation of types—namely, predefined and implementation-defined pragmas. However, such pragmas do not play an important role for the R1000 target. In particular, the predefined pragma Pack never needs to be specified, because packed representation of arrays and records is always in effect. Furthermore, the R1000 target supports no implementation-defined pragmas for controlling representation.

4.1. General Information about Scalar Object Size

Compilers on other computer systems typically represent the sizes of scalar objects in terms of the sizes of their base types. Because the sizes of base types are known at compile time, the representation of scalar object sizes is also determined statically, even if a dynamic range constraint is present. In contrast, the compiler on the R1000 represents scalar objects in terms of subtypes rather than base types. This choice has two consequences. First, a scalar object's size is generally smaller than that of its base type, because the R1000 compiler attempts to make subtypes as small as possible. Secondly, the representation of scalar objects whose subtypes have dynamic ranges is actually determined at run time, because this representation is not tied to the statically known sizes of base types.

4.2. Integer Types

The R1000 compiler supports two predefined integer base types:

```
type Integer      is range -2**31 + 1 .. 2**31 - 1;
type Long_Integer is range -2**63     .. 2**63 - 1;
```

Type declarations of the following form are implicitly derived from type Long_Integer (in this and the following examples, L stands for the low bound and H stands for the high bound):

```
type T is range L .. H;
```

Accordingly, the following are true:

```
T'Base'First = Long_Integer'First
T'Base'Last  = Long_Integer'Last
```

Note that L and H must fall within the bounds of Long_Integer; if they do not, the type declaration is rejected.

4.2.1. Representation of Integer Types

All integers are represented as two's complement binary numbers. The two predefined types, Integer and Long_Integer, have sizes of 32 and 64 bits, respectively. All arithmetic operations on integers are carried out using 64-bit arithmetic. Note that type Integer is symmetric about zero, but Long_Integer is not—Long_Integer has one more negative value than it has positive values, as allowed by LRM 3.5.4(7). Thus, Long_Integer uses all possible 64-bit values, whereas Integer uses all but one of the possible 32-bit values.

4.2.2. Minimum Sizes of Integer Types

The minimum possible size of an integer type is the minimum number of bits required for representing that type in the R1000 architecture. In general, the minimum size of an integer subtype or derived type is less than or equal to the minimum size of its parent type.

Minimum sizes are computed according to the following rules (these rules apply even if the expressions L and H are not static, in which case the sizes are computed at run time):

1. With constraints, subtypes and derived types are either the same size or smaller than the size of the parent type. Thus:

```
subtype S is  Integer range L .. H;   -- S'Size <= Integer'Size
type D is new S range L .. H;         -- D'Size <= S'Size <= Integer'Size
```

   In particular, the sizes of subtypes and derived types with constraints depend on the bounds of the specified range:

   a. If the range is null (that is, L > H), the size is zero. For example:

```
type T0 is range 4 .. 2;              -- T0'Size = 0
```

   b. If the range is unsigned (that is, $0 \le L \le H$), the size is the smallest integer S such that $H \le 2^S - 1$. For example:

```
type T1 is range 0 .. 7;              -- T1'Size = 3
type T2 is new Long_Integer
    range 0 .. Long_Integer'Last;     -- T2'Size = 63
```

   c. If the range is signed (that is, L < 0), the size is the smallest integer S such that $abs(H) \le 2^{S-1} - 1$ and $abs(L) \le 2^{S-1}$. For example:

```
type T3 is new Integer range -8 .. 15;        -- T3'Size = 5
```

2. Without constraints, subtypes and derived types are always the same size as their parent types. Thus:

```
subtype S is Integer;                 -- S'Size = Integer'Size
type D is new S;                      -- D'Size = S'Size = Integer'Size
```

4.2.3. Sizes of Integer Types

In the absence of an applicable size length clause, an integer subtype or derived type is represented using the minimum possible size, as defined by rules 1 and 2 in Section 4.2.2.

When an applicable size length clause is present, an integer subtype or derived type is represented using the size specified by the clause. The size specified in a size length clause must be greater than or equal to the minimum required size. Furthermore, the specified size must be less than or equal to 64. For example:

```
type T1 is range 0 .. 7;              -- T1'Size = 3
type T4 is new T1;
for T4'Size use 16;                   -- T4'Size = 16
```

Without the size length clause, the size of T4 (which has no constraints) would be 3, the size of its parent type. The size length clause specifies a value greater than the default size (3) and therefore overrides it. (A size length clause such as `for T4'Size use 2;` would be rejected.)

4.2.4. Sizes of Objects

An object of integer type has the same size as the subtype used to declare the object. For example:

```
type T1 is range 0 .. 7;              -- T1'Size = 3
X : T1;                               -- X'Size = 3
Y : T1 range 0 .. 3;                  -- Y'Size = 2, the size of
                                      -- the anonymous subtype
```

4.3. Enumeration Types

The representation of enumeration types can be controlled using enumeration representation clauses and size length clauses. The following sections describe the representation of enumeration types in the absence of these clauses as well as the implementation-dependent restrictions using these clauses.

4.3.1. Representation of Enumeration Types

Enumeration types correspond directly to integer types. This correspondence arises from the fact that each of the literals in an enumeration type is associated with an integer value. Consequently, the range of enumeration literals is represented as a range of integer values.

In the absence of an applicable enumeration representation clause, each enumeration literal in an enumeration type is represented by its position number. Thus, for an enumeration type with N enumeration literals, the literals are represented by the integers 0, 1, 2, ..., N – 1.

Enumeration representation clauses can be used to specify nondefault integer representations for the literals in enumeration types. For example, the enumeration representation clause below causes the literals A, B, and C to be represented by the integers 15, 16, and 101, respectively:

```
type T1 is (A, B, C);
for T1 use (15, 16, 101);
```

Enumeration representation clauses must specify integers in the range $-2^{31} + 1 .. 2^{31} - 1$.

4.3.2. Minimum Sizes of Enumeration Types

Because enumeration types are translated into ranges of integer values, the minimum size for an enumeration type is computed using rules 1 and 2 given in Section 4.2.2, above. When applying these rules to an enumeration type T, note that the low bound L stands for the integer representation of the first literal in the range (T'First) and the high bound H stands for the integer representation of the last literal in the range (T'Last). For example:

```
type T1 is (A, B, C);                -- T1'Size = 2 by rule 1.b.
                                     -- (L = 0 and H = 2)
subtype S is T1;                     -- S'Size = T1'Size by rule 2.

type T2 is new T1 range A..A;        -- T2'Size = 0, by rule 1.b.
                                     -- (L = 0 and H = 0)

type T3 is new T1;                                -- T3'Size = 32 by rule 1.c.
for T3 use (Integer'First, 0, Integer'Last);      -- (L = Integer'First and
                                                  --  H = Integer'Last)

type T4 is new T3 range B..C;        -- T4'Size = 31 by rule 1.b.
                                     -- (L = 0 and H = Integer'Last)
```

4.3.3. Sizes of Enumeration Types and Objects

In the absence of an applicable size length clause, an enumeration type is represented using the minimum possible size (see the previous section).

When an applicable size length clause is present, an enumeration type is represented using the size specified by the clause. As for integer types, the size specified in a size length clause must be greater than or equal to the minimum required size and must be less than or equal to 64. Following are examples of size length clauses used with enumeration types:

```
type T1 is (A, B, C);          -- (Minimum possible value of T1'Size is 2)
for T1'Size use 8;             -- T1'Size = 8

type T3 is new T1;                                -- (Minimum possible value
for T3 use (Integer'First, 0, Integer'Last);      -- of T3'Size is 32)
for T3'Size use 64;                               -- T3'Size = 64
```

4.4. Floating-Point Types

The R1000 compiler supports a single predefined floating-point base type:

```
type Float is digits 15
              range -1.79769313486231E+308 .. 1.79769313486231E+308;
```

A floating-point type declaration of the following form is implicitly derived from type Float:

```
type T is digits D [range L .. H];
```

Accordingly, the following are true:

```
T'Base'First  = Float'First
T'Base'Last   = Float'Last
T'Base'Digits = Float'Digits
```

4.4.1. Representation of Floating-Point Types

Values of type Float are represented using the double (64-bit) float format specified by the *IEEE Standard for Binary Floating Point Arithmetic* (ANSI/IEEE Std. 754-1985). The R1000 compiler does not support "not a number" values and the value negative zero. The encodings for these values are considered invalid.

The bit pattern for floating-point types consists of the following, arranged from left to right, for a total of 64 bits:

- 1 bit for the sign
- 11 bits for the exponent
- 52 bits for the mantissa

The sign bit is interpreted conventionally (0 is positive, 1 is negative).

The exponent bits are interpreted as an 11-bit unsigned integer biased by –1022. That is, if all 11 of the exponent bits are 0, the exponent is –1022; if all 11 of the exponent bits are 1, the exponent is 1025.

The interpretation of the mantissa bits depends on the value of the exponent:

- If the exponent equals the largest value (1025), then the value is "not a number," and the mantissa bits are not interpreted. Note that:
  - — No floating-point operation with valid operands will return such a value; instead, exceptions are raised wherever the IEEE standard specifies that "not a number" be returned.
  - — Floating-point operations with "not a number" values as operands yield unpredictable results. "Not a number" values can arise only through uninitialized variables or unchecked conversion.
- If the exponent equals the smallest value (–1022), then the mantissa bits are interpreted as a 52-bit unsigned integer with an implicit scaling factor of 2^{-52}. In other words, the mantissa is a multiple of 2^{-52} that lies in the range 0 .. 1 – 2^{-52}. This is the gradual underflow case.
- If the exponent is a value between –1022 and 1025, the mantissa bits are interpreted as if an extra bit (a 1) preceded the remaining 52. Accordingly, the mantissa bits are interpreted as a 53-bit unsigned integer with an implicit leading 1 (which normalizes the mantissa) and an implicit scaling factor of 2^{-53}. In other words, the mantissa is a multiple of 2^{-53} that lies in the range 0 .. 1 – 2^{-53}.

You can use the preceding information to convert bit patterns into mathematical values (and vice versa). For example, these steps convert the hexadecimal bit pattern 16#4008_0000_0000_0000# to the mathematical value 3.0:

1. Convert the first three hexadecimal digits (400) to binary notation: 2#0100_0000_0000#.
2. Obtain the sign from the binary notation. The sign is the leftmost bit, 0, which is positive.
3. Calculate the exponent from the remaining bits in the binary notation. To do this, evaluate the integer represented by these bits (1024) and subtract the bias: 1024 – 1022 = 2.
4. Calculate the mantissa from the remaining 13 digits of the hexadecimal notation. To do this:
   a. Insert the implicit leading 1:16#18_0000_0000_0000#.
   b. Evaluate the integer represented by this hexadecimal notation (24×2^{48}) and multiply by the scaling factor: $(24 \times 2^{48}) \times 2^{-53} = 0.75$.
5. Calculate the mathematical value of the entire bit pattern using the usual formula:

   $sign \times mantissa \times 2^{exponent}$

   Thus, the value is: $1 \times 0.75 \times 2^2 = 3.0$.

Table 3 lists various mathematical values and their floating-point representations.

Table 3 Floating-Point Representations of Mathematical Values

Mathematical Value	Floating-Point Representation	Comments
2 ** –1074	16#0000_0000_0000_0001#	Smallest positive representable number
(2 ** 1024) – (2 ** 971)	16#7FEF_FFFF_FFFF_FFFF#	Largest positive representable number (Standard.Float'Last)
2 ** 53	16#4340_0000_0000_0000#	Note: (2 ** 53) + 1 is the smallest positive *unrepresentable* integer
(2 ** 53) + 2	16#4340_0000_0000_0001#	
0.00	16#0000_0000_0000_0000#	
0.50	16#3FE0_0000_0000_0000#	
0.75	16#3FE8_0000_0000_0000#	
1.00	16#3FF0_0000_0000_0000#	
2.00	16#4000_0000_0000_0000#	
3.00	16#4008_0000_0000_0000#	
4.00	16#4010_0000_0000_0000#	
2 ** –63	16#3C00_0000_0000_0000#	System.Fine_Delta
–2 ** 63	16#C3E0_0000_0000_0000#	System.Min_Int
(2 ** 63) – 1	16#43DF_FFFF_FFFF_FFFF#	System.Max_Int (Note: same representation as next entry)
(2 ** 63) – (2 ** 10)	16#43DF_FFFF_FFFF_FFFF#	Largest representable number less than or equal to System.Max_Int

4.4.2. Size of Floating-Point Types and Objects

The size of all floating-point types, subtypes, and objects is 64 bits. A size length clause is allowed for a floating-point type only if it specifies a value of 64.

4.5. Fixed-Point Types

The R1000 compiler supports a set of anonymous predefined fixed-point base types of the following form:

```
type F is delta D range -(2.0**63) / S .. (2.0**63 - 1.0) / S;
for F'Small use S;
```

Exactly one such predefined fixed-point type exists for every value of S such that S is in the range 2.0^{-63} .. 2.0^{63} and can be represented as a rational number N/M (where $1 \leq N,M \leq 2^{63}$) *or* S is a power of two in the range 2.0^{-1025} .. 2.0^{960}. Common special cases for S are integers (for example, 2.0 or 10.0) and reciprocals of integers (for example, 0.5 or 0.1).

A fixed-point type declaration of the following form:

```
type T is delta D range L .. H;
for T'Small use S;
```

is implicitly derived from the anonymous predefined fixed-point type F for which F'Small = T'Small. The low bound L and the high bound H must be real numbers such that both L + S and H – S fall within the range F'First .. F'Last. If no such predefined base type exists, the type declaration is rejected.

A fixed-point type declaration T with no small length clause is implicitly derived from the predefined fixed-point base type F for which F'Small is the largest power of two that is not greater than T'Delta (LRM 3.5.9).

The R1000 compiler supports a single named predefined fixed-point type called Duration:

```
type Duration is delta 2.0**(-15) range -2.0**31 .. 2.0**31 - 1.0;
```

Note that Duration is not itself a base type but is a first-named subtype of the following anonymous predefined base type:

```
type Duration'Base is delta 2.0**(-15) range -2.0**48 .. 2.0**48 - 1.0;
for Duration'Base'Small use 2.0**(-15);
```

4.5.1. Representation of Fixed-Point Types

A value X of a fixed-point type T is represented as a multiple of T'Safe_Small. In particular, this multiple is the two's complement integer X/T'Safe_Small (if X is not divisible by T'Safe_Small, then the value X/T'Safe_Small is rounded to the nearest integer). For example, if X = 3.0 and T'Safe_Small = 0.5, then the value of X is represented by the integer 3.0/0.5 = 6. Note that in most cases, T'Small can be used instead of T'Safe_Small when determining integer representation of

fixed-point values. That is, T'Safe_Small = T'Small except where T is a subtype that has a different *delta* than its parent type.

In the absence of an applicable small length clause, the value of T'Small and the related value T'Safe_Small are determined by the rules given in LRM 3.5.9.

When an applicable small length clause is present, a fixed-point type is represented using the value of *small* specified by the clause. A small length clause is accepted only if it specifies a value S such that S is in the range 2.0^{-63} .. 2.0^{63} and can be represented as a rational number N/M (where $1 \leq N,M \leq 2^{63}$) *or* S is a power of two in the range 2.0^{-1025} .. 2.0^{960}. To guarantee the precision of results, the R1000 compiler currently requires that T'Mantissa ≤ 52 when the value specified in a small length clause is not a power of two. Until issues are resolved by the Ada Board or AJPO, the R1000 compiler does not support small length clauses on derived types that specify a value different from that of the parent type.

4.5.2. Minimum Sizes of Fixed-Point Types

There are two equivalent methods for calculating the minimum possible size of a fixed-point type. Both are presented below for completeness.

Method 1: The minimum possible size of a fixed-point type T can be calculated using the values for the attributes 'Mantissa, 'Small, and 'Safe_Small, defined in LRM 3.5.9:

- If type T has an unsigned range (that is, 0.0 ≤ T'First ≤ T'Last), the minimum size of type T is T'Mantissa * (T'Small/T'Safe_Small). For example:

```
type T1 is delta 1.0 range 0.0 .. 8.0;   -- T1'Safe_Small = T1'Small
                                         -- T1'Mantissa = 3
                                         -- Minimum size for T1 is 3
```

- If type T has a signed range (that is, T'First ≤ 0.0), the minimum size of type T is T'Mantissa * (T'Small/T'Safe_Small) + 1. For example:

```
type T2 is delta 0.5 range -8.0 .. 8.0;  -- T2'Safe_Small = T2'Small
                                         -- T2'Mantissa = 4
                                         -- Minimum size for T2 is 5
```

Method 2: Because fixed-point types are represented as integers, the minimum possible size of a fixed-point type can also be computed using the rules given in Section 4.2.2. Each fixed-point type T is represented using an integer subtype whose bounds are the integer representations of the model-number bounds for type T (see LRM 3.5.9). The model-number bounds and their integer representations are determined as follows:

- If type T has an unsigned range (that is, 0.0 ≤ T'First ≤ T'Last), the low model-number bound is 0.0 and the high model-number bound is –T'Large. Therefore, the integer representation of type T is as follows (assuming the integer conversion informally indicated by `int` has taken place) and the minimum size of type T is the size of this integer subtype:

```
type I is range 0 .. int(T'Large/T'Safe_Small);
```

For example, for type T1 declared below, the model numbers are multiples of 1.0 (T1'Safe_Small) in the range 0.0 to 7.0 (T1'Large). As shown, these values are represented using the integers in the range 0 .. 7, yielding an integer subtype whose minimum size is 3:

```
type T1 is delta 1.0 range 0.0 .. 8.0;    -- Minimum size for T1 is I'Size
type I is range 0 .. 7;                   -- I'Size = 3
```

- If type T has a signed range (that is, T'First ≤ 0.0), the high model-number bound is T'Large and the low model-number bound is –T'Large. Therefore, the integer representation of type T is as follows (assuming the integer conversions informally indicated by ***int*** have taken place) and the minimum size of type T is the size of this integer subtype:

```
type I is range int(-T'Large/T'Safe_Small) .. int(T'Large/T'Safe_Small);
```

For example, for type T2 declared below, the model numbers are multiples of 0.5 (T2'Safe_Small) in the range –7.5 (–T2'Large) to 7.5 (T2'Large). As shown, these values are represented using the integers in the range –15 .. 15, which yields an integer subtype whose minimum size is 5:

```
type T2 is delta 0.5 range -8.0 .. 8.0;   -- Minimum size for T2 is I'Size
type I is range -15 ..15;                 -- I'Size is 5
```

4.5.3. Sizes of Fixed-Point Types and Objects

In the absence of a size length clause, a fixed-point type T is represented using either the minimum possible size or a size that is one bit larger than the minimum. The minimum possible size is used only when it allows the specified bounds (T'First and T'Last) to be represented. However, as shown in the previous section, the minimum possible size is evaluated using model-number bounds rather than the specified bounds. Because the model-number bound may fall within the specified bounds (for example, T'Large may be slightly less than T'Last), the minimum possible size may not be large enough to allow the representation of the specified bounds. In such cases, the R1000 compiler chooses a size that is one bit larger than the minimum to accommodate the specified bounds (that is, T'Last ≤ T'Safe_Large). More specifically, the R1000 compiler evaluates the size of a fixed-point type T by calculating the size of an integer subtype of the following form (assuming the integer conversions informally indicated by ***int*** have taken place):

```
type I is range int(T'First/T'Safe_Small) .. int(T'Last/T'Safe_Small);
```

The resulting size must be less than or equal to 64 bits.

For example, consider the type T1, whose minimum possible size is 3, as shown above. This minimum size derives from a representation based on the model numbers 0.0 and 7.0 (T'Large). However, this representation does not include 8.0 (T'Last). Therefore, the R1000 compiler chooses a size that is one bit larger than the minimum and that can include the specified bounds:

```
type T1 is delta 1.0 range 0.0 .. 8.0     -- Minimum size is 3
                                          -- T1'Size = 4
```

In contrast, consider the type T3, whose minimum possible size is also 3. Because its model-number bounds correspond to the specified bounds, the R1000 compiler can (and does) choose the minimum possible size for T3'Size:

```
type T3 is delta 1.0 range 0.0 .. 7.0      -- Minimum size is 3
                                           -- T3'Size = 3
```

When an applicable size length clause is present, a fixed-point type is represented using the specified size. The size specified in a size length clause must be greater than or equal to the minimum possible size. (Thus, the specified size may in some cases be one bit smaller than the size chosen by the R1000 compiler.) The specified size must also be less than or equal to 64 bits.

For example, consider T4, a derived type whose parent type is T1 given above. Without the size length clause, the R1000 compiler would choose a representation of four bits; however, because a size length clause is specified, this choice is overridden and the R1000 compiler is forced to represent type T4 using the minimum possible size of three bits:

```
type T4 is new T1;                         -- Minimum size is 3
for T4'Size use 3;                         -- T4'Size = 3
```

4.6. Access Types

The representation of access types can be controlled using size length clauses and storage-size length clauses. The following sections describe the representation of access types in the absence of these clauses as well as the implementation-dependent restrictions using these clauses.

4.6.1. Representation of Access Types

For each access type, a contiguous block of memory called a *collection* is reserved to provide storage for allocated objects of the designated type. A collection for a given access type comes into existence at the time the access type's declaration is elaborated. If insufficient memory is available at the time such a declaration is elaborated, the Storage_Error exception is raised.

Each object of an access type contains a value that points to an object of the designated type within the access type's collection. Each such value is the *bit offset* (the number of bits) between the beginning of the access type's collection and the beginning of the designated object. In this way, the R1000 compiler differs from many other compilers, for which the values of access types are represented as absolute machine addresses.

4.6.2. T'Size and T'Storage_Size

For an access type T of the following form (where the ellipsis stands for the name of the designated type):

```
type T is access ... ;
```

there are two relevant measures of size:

- T'Size is the number of bits used to represent the objects of access type T. Thus, T'Size is the number of bits in which bit offset values are represented.
- T'Storage_Size is the size (in bits) of the collection that is reserved for T. Note that by measuring the collection size in bits, the R1000 compiler differs from many other compilers, which measure collection size in bytes.

As described in the LRM, values for T'Size and T'Storage_Size can be specified using size length clauses and storage-size length clauses, respectively. Whereas size length clauses specify the precise size to be used for access objects, a storage-size length clause merely specifies the minimum size to be used for an access type's collection. In fact, the R1000 compiler uses the specified storage-size value only when this value is a power of two; otherwise, the R1000 compiler rounds this value up to the next power of two. For example:

```
type T is access ... ;
for T'Storage_Size use 1000;          -- T'Storage_Size = 1024
```

4.6.3. Restrictions on Size and Storage-Size Specifications

If both a size length clause and a storage-size length clause are specified for an access type T, then the values specified in these clauses must satisfy all of the following restrictions:

- The value specified for T'Size is in the range 1 .. 32. This value must be static.
- The value specified for T'Storage_Size is in the range 2 .. 2^{32}. This value may be static or dynamic.
- The value specified for T'Storage_Size is less than or equal to $2^{T'Size}$. (Thus, T'Storage_Size establishes a minimum value for T'Size; conversely, T'Size establishes a maximum value for T'Storage_Size.)

Failure to satisfy these restrictions results in the errors listed below:

- If the first restriction is not satisfied (that is, T'Size is out of range), the size length clause is rejected at compile time.
- If the second restriction is not satisfied (that is, T'Storage_Size is out of range) and the value specified for T'Storage_Size is static, the storage-size length clause is rejected at compile time. If, however, the value of T'Storage_Size is both out of range and dynamic, the Constraint_Error exception is raised at run time.
- If the third restriction is not satisfied and the value specified for T'Storage_Size is static, then the order of the size and storage-size length clauses is important—the first of the two clauses is accepted and the second clause is rejected at compile time. If, however, the value of T'Storage_Size is dynamic, the Storage_Error exception is raised at run time.

Although all values for T'Size and T'Storage_Size are accepted if they fall within the ranges given above, certain small values are not useful in practice. In particular, if T'Size ≤ 7 and T'Storage_Size ≤ 128, the collection of access type T is too small for objects of the designated type to be allocated. This is because the R1000 compiler allocates a header of at least 128 bits at the beginning of every collection. The header requires more than 128 bits when unchecked deallocation is enabled.

Note that the R1000 compiler allows size length clauses to apply to derived access types; however, the size specified for a derived type must be the same as the size of its parent type.

4.6.4. Compiler-Chosen Representations

When neither a size length clause nor a storage-size length clause applies to an access type T, the R1000 compiler chooses the following representation for T:

- The value of T'Size is 24
- The value of T'Storage_Size is 2^{24}

When a length clause is specified for one of T'Size or T'Storage_Size (but not both), the R1000 compiler chooses the remaining value in accordance with the restrictions given above. Thus:

- If a size length clause is specified, but a storage-size length clause is not, the R1000 compiler sets T'Storage_Size = $2^{T'Size}$. For example:

```
type T is access ...;
for T'Size use 20;                          -- T'Storage_Size = 2**20
```

- If a storage-size length clause is specified, but a size length clause is not, several cases must be considered:

  — If the value specified for T'Storage_Size is static, then the R1000 compiler sets T'Size = $\lceil \log_2 T'Storage\_Size \rceil$. This value is the least integer that is greater than or equal to $\log_2 T'Storage\_Size$. For example:

```
type T is access ...;                       -- T'Storage_Size = 1024
for T'Storage_Size use 1000;                -- T'Size = 10
```

  — If the value specified for T'Storage_Size is dynamic and the storage-size length clause immediately follows the access-type declaration, then the R1000 compiler sets T'Size = $\lceil \log_2 T'Storage\_Size \rceil$ as before; now, however, this value is computed at run time:

```
type T is access ...;                       -- T'Size = ceiling(log2f(x))
for T'Storage_Size use f(x);                -- computed at run time
```

  — If the value specified for T'Storage_Size is dynamic and the storage-size length clause does *not* immediately follow the access-type declaration, then the R1000 compiler sets T'Size = 24:

```
type T is access ...;                       -- T'Size = 24
...
other declarations
...
for T'Storage_Size use f(x);
```

4.7. Task Types

Each object of a task type is associated with a task control block that contains all the information the system needs to schedule the task. A given task object is represented as the machine address of the task control block associated with it. Because tasks are represented as machine addresses, all task types, subtypes, and objects are the size of System.Address, which is 64. Size length clauses are allowed only if they specify a size of 64.

In the absence of an applicable storage-size length clause, the storage-size for a task type, subtype, or object is 2^{32}. A storage-size length clause can be provided to assign a different value to T'Storage_Size; however, such a clause is primarily for compatibility with other compilers and has no real effect on the R1000 compiler. The value specified in a storage-size length clause may be static or dynamic and must be in the range 0 .. 2^{32}.

4.8. Array Types

The elements of an array are stored in contiguous memory locations with the first array component stored at the lowest address. All components of an array have the same size, with no gaps between them (in essence, pragma Pack is always in effect). For example, one-dimensional arrays of type A1 are laid out as shown in Figure 1:

```
type A1 is array (1 .. N) of T;
```

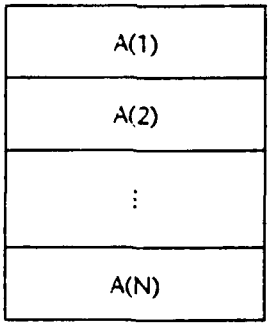


Figure 1 A One-Dimensional Array

Similarly, two-dimensional arrays of type A2 have the layout shown in Figure 2:

```
type A2 is array (1 .. N, 1 .. M) of T;
```

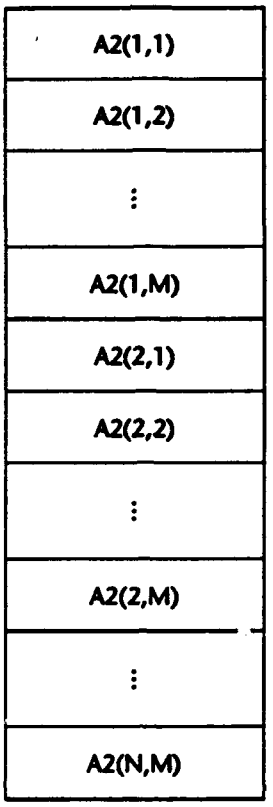


Figure 2 A Two-Dimensional Array

4.8.1. Dope Vectors

Every array has associated with it information about its size and the bounds of each of its dimensions. This information is represented in the form of a *dope vector* (sometimes called an *array descriptor*). For all arrays whose types are constrained and for most arrays whose types are unconstrained, the dope vector is stored as a separate object associated with the array's type or subtype. When this is the case, the dope vector is an internal construct that has no consequences for the representation of the array itself.

However, when an array of an unconstrained type is used in either of the following two ways, the array's dope vector is included as part of the array's representation:

- The array's type is the designated type of an access type. For example:

```
type S is access String;          -- String is an unconstrained array type
X : S := new String (1 .. 17);    -- Allocate a string with bounds 1..17
Y : S := new String (1 .. 36);    -- Allocate a string with bounds 1..36
```

  As shown, each of the arrays allocated in the access type's collection may have different bounds. Consequently, the dope vector containing information about the array's bounds must

be stored with the array itself. Extra space is allocated with each array to accommodate its dope vector.

- The array is a component of a record and is constrained by a discriminant of that record. For example:

```
type R (D : Integer) is
    record
        F : String (1 .. D);        -- String is an unconstrained array type
    end record;
X : R(17);                          -- String component with bounds 1..17
Y : R(36);                          -- String component with bounds 1..36
```

  As shown, each object of type R may have a string component with different bounds. Consequently, the dope vector containing information about the string component's bounds is stored with the string itself. Extra space is allocated with each array field to accommodate its dope vector.

The dope vector for an array contains bounds and size information for each of its dimensions. For all but the last dimension, this information is represented in 96-bit triples. Thus, if an array A has N dimensions, the triple for a given dimension K (K < N) contains:

- The value of A'First(K), stored in the first 32 bits of the triple
- The value of A'Length(K), stored in the second 32 bits of the triple
- The size (in bits) of the subarray formed by dimensions K+1 .. N, stored in the remaining 32 bits of the triple. The informal notation A'Size(K) is used to express this subarray size.

The last dimension is represented as a 64-bit pair containing the first two of the three values listed above (that is, A'First(N) and A'Length(N)). The subarray size is omitted because the last dimension defines no subarray.

If A is a nonnull array (an array that contains at least one element), its dope vector is exactly as described above; thus the size of the dope vector for a nonnull array is 96N – 32 bits.

If A is a null array (an array that contains zero elements because at least one dimension has a null range), its dope vector includes an additional series of 32-bit entries that contain the values of A'Last for each dimension. Thus, the size of a dope vector for a null array is 128N – 32 bits.

Figure 3 shows the general dope-vector layout for N-dimensional nonnull and null arrays:

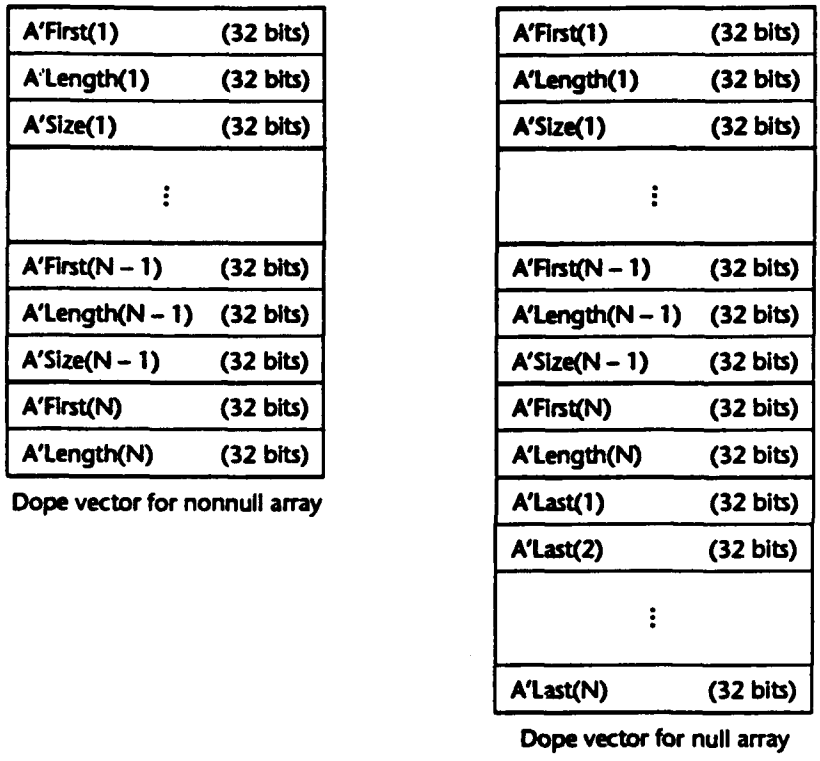


Figure 3 General Dope-Vector Layout for Nonnull and Null Arrays

For example, the following declarations show that a one-dimensional, nonnull array of type A3 has been allocated in the collection of access type P1. Figure 4 shows the array's layout:

```
type A3 is array (Positive range <>) of T;
type P1 is access A3;
X : P1 := new A3 (1 .. N);
```

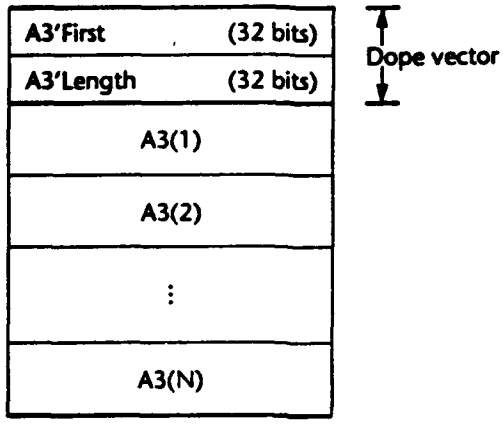


Figure 4 A One-Dimensional Array with a Dope Vector

Similarly, the following declarations show that a two-dimensional, nonnull array of type A4 has been allocated in the collection of access type P2. Figure 5 shows the array's layout:

```
type A4 is array (Positive range <>, Positive range <>) of T;
type P2 is access A4;
Y : P2 := new A4 (1 .. N, 1 .. M);
```

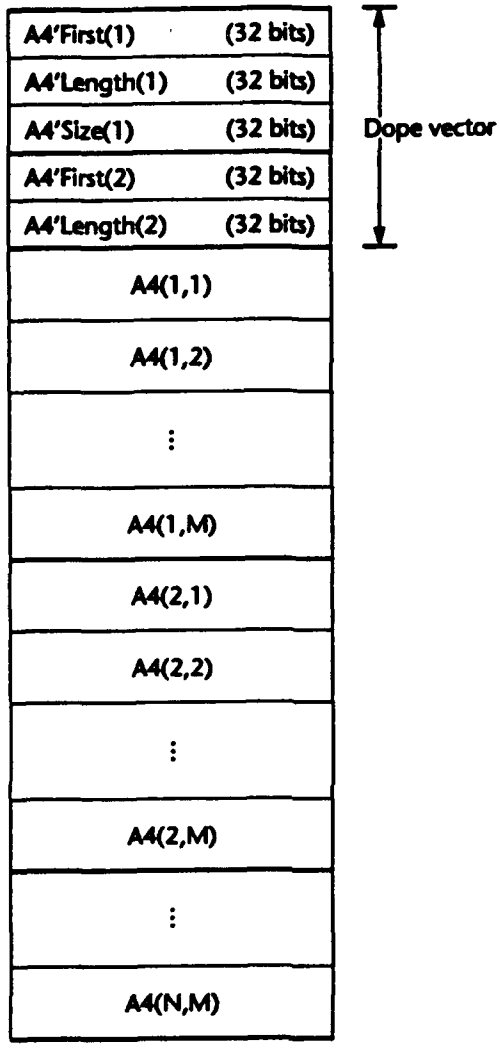


Figure 5 A Two-Dimensional Array with a Dope Vector

4.8.2. Sizes of Array Types and Subtypes

The size of a constrained array type is obtained by multiplying the number of components by the size of the components. For example, the following constrained array type T1 has 10 Boolean components, and the size of each component is Boolean'Size = 1. Therefore, the size of the array type T1 is 10 × 1 = 10:

```
type T1 is array (1..10) of Boolean;                -- T1'Size = 10
```

Obtaining the size of an unconstrained array type involves multiplying the maximum number of components by the size of the components and then adding to this product the size of the dope vector. The dope-vector size is added because the size of an unconstrained array type must be large enough to accommodate any array object, including one whose representation contains a

dope vector (see Section 4.8.1, above). Furthermore, because dope-vector sizes differ for nonnull and null arrays, the size of an unconstrained type must be able to accommodate either kind of array. Therefore, the formula for calculating the size of an unconstrained N-dimensional array type is given informally as:

Max (maximum_number_of_components × component_size, 32N) + 96N – 32

This formula ensures that the minimum size of the unconstrained array type is 32N + 96N – 32 = 128N – 32, which is the size required for the dope vector for a null array.

For example, the following unconstrained array type T2 has a maximum of two integer components; the size of each component is Integer'Size = 32. Because the product 2 × 32 is greater than 32N (which equals 32 for a one-dimensional array), the size of T2 is (2 × 32) + (96 × 1) – 32 = 128:

```
type T2 is array (Boolean range <>) of Integer;  -- T2'Size = 128
```

Now consider the following unconstrained array type T3, which has a maximum of two Boolean components, each of size Boolean'Size = 1. Because the product 2 × 1 is less than 32N (which again equals 32), the size of T3 is 32 + (96 × 1) – 32 = 96:

```
type T3 is array (Boolean range <>) of Boolean;  -- T3'Size = 96
```

Note that the size of an array subtype is less than or equal to the size of its base type. In particular, a constrained subtype is smaller than its base type if the base type is unconstrained. This is because the size of the unconstrained base type is calculated using the maximum possible number of components and also includes the size of the dope vector. In contrast, the size of the constrained subtype is calculated using the specified number of components (which may be less than the maximum) and does not include the size of the dope vector.

The size of an array type or subtype cannot be changed. Thus, a size length clause is accepted only if it specifies the size that would normally be used by the R1000 compiler.

4.8.3. Sizes of Arrays

All arrays are objects of some constrained subtype and, in most cases, are of the same size as that subtype. However, in two cases, the size of an array will be larger than the size of its constrained subtype—namely, when the array is a designated object of an access type or when the array is a record component that is constrained by a discriminant. In each of these cases, a dope vector is included in the representation of the array. Therefore, the size of the array is equal to the size of its subtype plus the size of the dope vector. (Recall that the dope vector size is 96N – 32 for an N-dimensional nonnull array, and 128N – 32 for an N-dimensional null array.)

For example, consider the following declarations, in which the unconstrained array type String is used as the designated type of an access type S:

```
type S is access String;
Z : S := new String (1 .. 10);          -- Z.all'Size = 144
```

The access object Z allocates an array of a constrained subtype of String with 10 components. Because the components are of type Character, their size is Character'Size = 8, so the size of the constrained subtype is 10 × 8 = 80. Because the allocated array is one-dimensional and nonnull, its dope-vector size is 96 – 32 = 64, so the size of the allocated array is 80 + 64 = 144.

4.9. Record Types

The representation of a record type is divided into fields corresponding to the record's various components. In the representations of certain record types, the set of component fields is prefixed with either one or two compiler-generated fields. The layout and sizes of record representation fields are described in following sections.

4.9.1. Layout of Fields Representing Record Components

Record types may have any combination of several different kinds of components. *Fixed components* refer to any components that are declared in the fixed part of the declaration; *variant components* refer to any components that are declared in the variant part of the declaration. Furthermore, a given fixed or variant component is either *indirect* (an array or record whose constraint is dependent on a discriminant) or *direct* (all other components). *Discriminants* are the components on which the values of other components may depend.

The fields that represent a record type's components are arranged by component kind and occur in the following order, with no gaps between them (in essence, pragma Pack is always in effect):

- Fields for discriminants
- Fields for fixed components (within the fixed part, fields for direct components precede fields for indirect components)
- Fields for variant components (within each variant of the variant part, fields for direct components precede the fields for indirect components)

Where there are multiple components of a given kind (for example, multiple discriminants or fixed direct components), the order of their fields is taken from the Ada source code.

A record whose type or subtype contains a variant part can have exactly one *active variant* at a time, where an active variant is the particular variant that is selected by the current value of the discriminant. Accordingly, the representation of a variant record with a given discriminant value contains fields for only one variant (namely, the active variant). The active variant's fields immediately follow the fields for the record's fixed components. Thus, when two variant records of the same type have different discriminants, the representations of these records are identical up to the fields for the active variant components.

Because the value of an indirect component depends on the value of the discriminant, the size of the indirect component's value may differ from one record to another. Therefore, an indirect component is represented using two separate fields. The first field (called an *indirect pointer field* is a 64-bit pointer to the second field (called a *data field*), which contains the actual value of the component. The indirect pointer field consists of:

- A 32-bit offset, which is the integer number of bits from the beginning of the record and the data field
- A 32-bit integer expressing the size of the data field

Within a given record, the data fields for all indirect components (both fixed and variant) occur at the end of the record, following the representation of the active variant (if any).

4.9.2. Layout of Compiler-Generated Fields

For discriminated record types (including variant record types), the compiler generates the first field in the representation. This field consists of a single bit called the *constrained object bit*. The constrained object bit, which corresponds to the 'Constrained attribute, is used as part of the memory-protection scheme of the R1000 architecture. The constrained object bit precedes the fields for the user-defined components.

For variant record types with a nonempty variant part, the compiler generates an additional field that follows the constrained object bit. This second field consists of an 8-bit *variant clause index* (VCI), which is used for fast discriminant checking whenever a component in a variant is referenced. That is, within a given variant record type R, each variant in the variant part is implicitly assigned a number, starting with 1. (The variants are numbered sequentially in the order in which they appear in the type's declaration.) When a particular record of type R selects an active variant, the number assigned to that variant is then stored in the record's VCI field.

If a variant itself contains a nonempty variant part, then the variant is represented as an anonymous record, prefixed with its own constrained object bit and VCI field.

4.9.3. Examples of Record Layout

Assume that type R1 is a record type containing only fixed direct components:

```
type R1 is
    record
        F1 : Integer;                    -- fixed direct component
        F2 : Boolean;                    -- fixed direct component
        F3 : String (1 .. 10);           -- fixed direct component
    end record;
```

Then a record of type R1 has the representation shown in Figure 6 (note that the size of each field is determined by the component subtype):

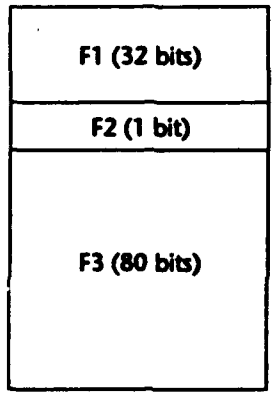


Figure 6 A Record with Fixed Direct Components

Assume that type R2 is a discriminated record type with one fixed direct component and two fixed indirect components:

```
type R2 (D : Integer) is                    -- discriminant component
    record
        F1 : Integer;                       -- fixed direct component
        F2 : String (1 .. D);               -- fixed indirect component
        F3 : String (1 .. D);               -- fixed indirect component
        F4 : Boolean;                       -- fixed direct component
    end record;
```

Then a record of type R2 has the representation shown in Figure 7. As shown, the representation of R2 has a constrained object bit, followed by the fields for the discriminant D and the direct components F1 and F4, followed by pointer and data fields for the indirect components F2 and F3. Note that the sizes of the discriminant field and the direct fields are determined by the component subtypes, whereas the sizes of the data fields depend on the discriminant value.

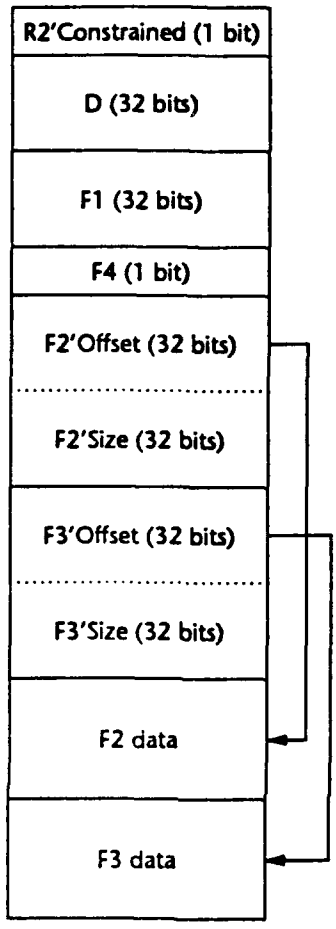


Figure 7 A Record with Direct and Indirect Components

Assume that type R3 is a variant record type with one discriminant, one fixed direct component, and a variant part containing two variants. These variants, which each contain a single direct component, are assigned VCI numbers 1 and 2:

```
type R3 (D : Boolean) is                    -- discriminant component
    record
        F1 : Integer;                       -- fixed direct component
        case D is
            when True =>                    -- VCI is 1
                F2 : Integer;               -- variant direct component
            when False =>                   -- VCI is 2
                F3 : Float;                 -- variant direct component
        end case;
    end record;
```

Depending on the value of the discriminant, a record of type R3 has either of the representations shown in Figure 8. Both representations begin with the constrained object bit and a VCI field, followed by the fields for the discriminant D and the fixed direct component F1. When the

record's discriminant is True, the representation on the left is used, in which the VCI is 1 and the active variant is one containing the Integer component. When the record's discriminant is False, the representation on the right is used, in which the VCI is 2 and the active variant is one containing the Float component.

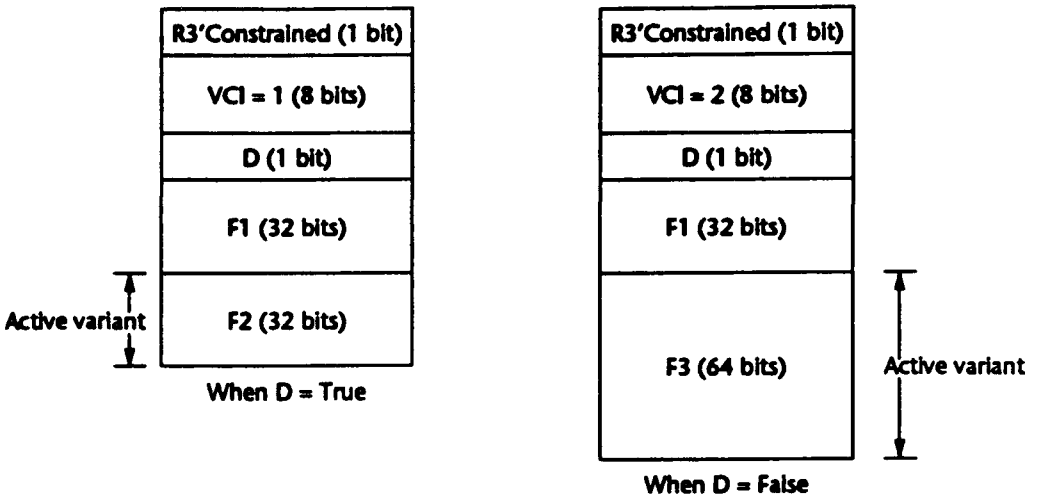


Figure 8 A Record with Fixed and Variant Components

4.9.4. Sizes of Record Types and Subtypes

In general, the size of a record type is the sum of the sizes of the fields that represent its components *plus* the sizes of any compiler-generated fields. For a variant record type, the sum includes the size of the active variant if the discriminant is constrained; otherwise, the size of the largest variant is used. The following paragraphs discuss in more detail how sizes are calculated for particular kinds of record types. (Table 4, in Section 4.9.6, summarizes the sizes for the various kinds of fields.)

1. *Nonvariant Record Types:* Nonvariant record types have a fixed part but no variant part. The sizes of such types are affected by whether or not there is a discriminant:

   a. A nonvariant record type with no discriminant has only fixed direct components. Therefore, the size of the type is simply the sum of the sizes of all of the component subtypes. For example, the size of type R1 (presented in Section 4.9.3, above) is the sum of the sizes of Integer, Boolean, and a string with 10 characters.

   b. A nonvariant record type with a discriminant may have both fixed direct and fixed indirect components. Therefore, the size of the type is the sum of the sizes of the component subtypes *plus* the size of the constrained object bit (1 bit) *plus* the sizes of any indirect field pointers (64 bits per pointer). Recall from Section 4.8 that if any component contains a discriminant-dependent array, the size of that component subtype must contain the size of the array's dope vector.

2. *Variant Record Types:* Variant record types have both a discriminant and a variant part. The sizes of such types are affected by whether or not the discriminant is constrained (a constrained discriminant has a specific value, which selects an active variant):

   a. A variant record with a constrained discriminant has exactly one active variant in addition to zero or more fixed direct and indirect components. Therefore, the size of the type is the size of its fixed part (as given in item 1b above) *plus* the size of the VCI field (8 bits) *plus* the size of the active variant. The size of the active variant is, in turn, the sum of the sizes of its component subtypes and their dope vectors (if any), *plus* any indirect pointers, *plus* (if the variant contains a nested variant part) the size of the nested active variant, including its VCI field and constrained object bit.

   b. A variant record with an unconstrained discriminant has multiple variants, each of which is a candidate for selection as the active variant. Therefore, the size of the type is the size of its fixed part (as given in item 1b above) *plus* the size of the VCI field (8 bits) *plus* the size of the largest variant. (Thus, you must evaluate the size of each variant and use the maximum of these sizes.) The size of a variant is the sum of the sizes of its component subtypes and their dope vectors (if any), *plus* any indirect pointers, *plus* (if the variant contains a nested variant part) the size of the largest nested variant, including its VCI field and constrained object bit.

The R1000 compiler chooses the minimum size for a record type, given the record layout described above. The size of a record type cannot be increased using a size length clause. A size length clause is accepted only if it specifies the compiler-chosen size for the type.

4.9.5. Sizes of Record Objects

The size of a record object is the size of its subtype.

4.9.6. Summary of Record Field Sizes

Table 4 summarizes the sizes of computer-generated fields as well as fields that represent various kinds of components.

Table 4 Summary of Record Field Sizes

Field	Size
Constrained object bit	1 bit
Variant clause index (VCI)	8 bits
Discriminant	Size of component subtype
Direct component	Size of component subtype
Indirect component (record)	64 bits for pointer + size of component subtype
Indirect component (array)	64 bits for pointer + size of component subtype (including dope-vector size)

4.9.7. Record Representation Clauses

Record representation clauses may be used to force a different field order, to introduce gaps between fields, or to alter the size of a field. However the following restrictions must be observed:

- The compiler-generated fields (constrained object bit and VCI), if present, cannot be moved from their normal positions at the beginning of the record.
- The discriminant fields may not be reordered, nor may gaps be introduced between discriminants. The size of a discriminant may be changed.
- If the field order is changed from the default, the fields must still be ordered by component kind:
  - — Discriminant
  - — Fixed components (direct preceding indirect)
  - — Variant components (direct preceding indirect within each variant)
- The size of fields for components of subtype array, record, access, or task cannot be changed from their default values. The size of other fields must follow the rules for size length clauses for those types.
- The placement of indirect field pointers and indirect data fields cannot be specified using a record representation clause.
- The constraints on the component *subtype, if any, must* be static.

5. IMPLEMENTATION-GENERATED NAMES

No implementation-generated names are used by the R1000 compiler.

6. ADDRESS CLAUSES

The R1000 compiler uses a memory-protection scheme that prohibits arbitrary address computations including the use of an address literal. For this reason, the type System.Address is a private type and no conversion functions are provided to convert other values to System.Address. Consequently, address clauses are not supported by the R1000 compiler.

6.1. Address Clauses for Interrupt Entries

Because interrupts do not exist in the R1000 architecture, address clauses for interrupt entries are not needed and thus are not supported.

7. UNCHECKED PROGRAMMING

7.1. Unchecked_Conversion Function

The Unchecked_Conversion generic function converts objects of one type to objects of another type. Its formal parameter list is:

```
generic
  type Source is limited private;
  type Target is limited private;
function Unchecked_Conversion (S : Source) return Target;
```

where Source specifies the original type of the object to be converted and Target specifies the type that will result from the conversion. More specifically, when a source object S is converted, its bit pattern (binary representation) is converted to the bit pattern of the Target type.

The source and target bit patterns are left-justified. Thus, the leftmost bit of the source object becomes the leftmost bit of the target object. If the source object is smaller than the target object, the target object will contain an undefined pattern in the bit positions not filled by the source. If the source object is larger than the target object, then the rightmost bits of the source object are ignored.

Unchecked conversion from a Source to a Target type yields unpredictable results and therefore is not recommended if:

- Either type is or contains an unconstrained array type
- Either type is or contains a discriminated record

This is because the bit patterns of unconstrained array types and discriminated records contain compiler-generated fields such as dope vectors (see Section 4.8.1) and constrained object bits, variant clause indexes, or indirect field pointers (see Section 4.9). Conversion *from* one of these types may cause the target data to contain unexpected bits. Conversion *to* one of these types may cause bits from the source object to be mapped into bit positions in the target that are reserved for compiler-generated fields, possibly causing these fields to contain illegal values. When this happens, an exception such as System.Type_Error, System.Capability_Error, or System.Assertion-_Error is raised at run time. Note that conversions involving constrained array subtypes or records without discriminants produce reliable results.

A faster alternative to the Unchecked_Conversion function is package Unchecked_Conversions. For additional information and examples, see the reference entries for the Unchecked_Conversion function and package Unchecked_Conversions in the Programming Tools (PT) book of the *Rational Environment Reference Manual.*

7.2. Unchecked_Deallocation Procedure

The Unchecked_Deallocation procedure may be instantiated for any access type. Its purpose is to deallocate storage for the designated objects, so that the storage can be reclaimed and reused. This procedure can reclaim storage for an access type only if deallocation has already been enabled in either of the following ways:

- The user has included pragma Enable_Deallocation for the access type, *or*

- The applicable library switch R1000_Cg.Enable_Deallocation had the value True when the program was compiled, and there is no pragma Disable_Deallocation for the access type.

The formal parameter list of the Unchecked_Deallocation procedure is:

```
generic
  type Object is limited private;
  type Name   is access  Object;
procedure Unchecked_Deallocation (X : in out Name);
```

If deallocation has been enabled for the access type in question, the Unchecked_Deallocation procedure first reclaims the storage for the object designated by the pointer X and then assigns null to X. If deallocation has not been enabled for the access type, then storage is not reclaimed, although X is still set to null. Furthermore, even if deallocation is enabled, storage is reclaimed only if it is safe. For example, if an access type has a designated type that is or contains a task, the R1000 architecture prevents storage reclamation for that access type because such reclamation could jeopardize system integrity. In these situations, X is set to null even though storage is not reclaimed. Note that the rules for determining whether it is safe to reclaim storage are complex. The user can determine whether storage reclamation is possible through an instantiation of the generic function Allows_Deallocation.

Enabling deallocation for an access type T causes each allocated object to consume additional space (specifically, twice T'Size) within the collection. In the absence of size length clauses or storage-size length clauses, T'Size is 24 bits; consequently, each allocated object typically consumes an additional 48 bits of the collection. This additional space is used for a free list, which serves to keep track of the portions of the collection that are in use. Furthermore, enabling deallocation increases the size of the collection header, causing it to be greater than 128. Thus, enabling deallocation has the effect of decreasing the maximum number of objects that can be allocated in a collection. For this reason, deallocation is normally left disabled in a library and enabled only when needed, either on a library-by-library basis (with the R1000_Cg.Enable_Deallocation switch) or on a type-by-type basis (with pragma Enable_Deallocation).

For additional information, see the reference entries for the Allows_Deallocation function and the Unchecked_Deallocation procedure in the Programming Tools (PT) book of the *Rational Environment Reference Manual*.

8. INPUT/OUTPUT PACKAGES

The Environment supports all of the I/O packages defined in Chapter 14 of the LRM, except for package Low_Level_Io, which is not needed. The Environment also provides a number of other I/O packages. The packages defined in Chapter 14, as well as the other I/O packages supported by the Environment, are more fully documented in the Text Input/Output (TIO) and Data and Device Input/Output (DIO) books of the *Rational Environment Reference Manual*. The following subsections summarize the implementation-dependent features of the Chapter 14 I/O packages.

8.1. Filenames

Filenames must conform to the syntax of Ada identifiers. They can, however, be keywords of the Ada language.

8.2. Form Parameter

The Open and Create procedures in packages Text_Io and Io each have a Form parameter for controlling the way in which files, terminals, and Ada units are read. Form parameters are similar to Options parameters in other Environment commands and can be specified using the same syntax. The particular form options that are available depend on what is being read.

The following form option is available when reading Ada units:

- **`Page_Pragma_Mapping`**

  A Boolean option. Specifies whether instances of pragma Page cause an ASCII.FF to be inserted at the end of the line to force a page break. The default is False.

The following form options are available for terminal-specific I/O:

- **Crlf**

  A Boolean option. Specifies whether ASCII.LF should be mapped to the combination ASCII.CR and ASCII.LF. The default is True.

- **Echo**

  A Boolean option. Specifies whether to echo input. The default is True.

- **Editing=*literal***

  Specifies whether line editing is in effect. The literal value Line causes line editing to be in effect. The literal value None disables line editing. The default value is Line.

For text files:

- **Gateway**

  A Boolean option. Specifies whether a file is to be read as a gateway file (a category of remote files that are accessed through specific Rational layered products). The default is False.

- **Synchronized**

  A Boolean option. Specifies whether the read operation requires a complete and consistent copy of the file. When False, unsynchronized access is permitted, so that files can be read even if they are open for editing or are being written by an active program. When True, files can be read only if they are closed. The default is False.

8.3. Instantiations of Packages Direct_Io and Sequential_Io

Instantiations of packages Direct_Io and Sequential_Io with access types are permitted. However, if files are created or opened using such instantiations, the Use_Error exception is raised.

8.4. Count Type

The Count type for package Text_Io and package Direct_Io is defined as:

```
package Text_Io is
  ...
  type Count is range 0 .. 1_000_000_000;
  ...
end Text_Io;

package Direct_Io is
  ...
  type Count is new Integer
                range 0 .. Integer'Last/Element_Type'Size;
  ...
end Direct_Io;
```

8.5. Field Subtype

The Field subtype for package Text_Io is defined as:

```
subtype Field is Integer range 0 .. Integer'Last;
```

8.6. Standard_Input and Standard_Output Files

When a job is run from a command window, these files are the interactive input/output windows provided by the Rational editor. When a job is run from package Program, options allow the user to specify what Standard_Input and Standard_Output will be.

8.7. Internal and External Files

More than one internal file can be associated with a single external file for input only. Only one internal file can be associated with a single external file for output or in-out operations.

8.8. Sequential_Io and Direct_Io Packages

Package Sequential_Io can be instantiated for unconstrained array types or for types with discriminants without default discriminant values. Package Direct_Io cannot be instantiated for unconstrained array types or for types with discriminants *without* default discriminant values.

8.9. Terminators

The line terminator is denoted by the character ASCII.LF, the page terminator is denoted by the character ASCII.FF, and the end-of-file terminator is implicit at the end of the file. A line terminator directly followed by a page terminator is compressed to the single character ASCII.FF. The line and page terminators preceding the file terminator are implicit and do not appear as characters in the file. For the sake of portability, programs should not depend on this representation, although it can be necessary to use this representation when importing source files from another environment or exporting source from the Rational Environment.

8.10. Treatment of Control Characters

Control characters, other than the terminators described above, are passed directly to and from files to application programs.

8.11. Concurrent Properties

The Chapter 14 I/O packages assume that concurrent requests for I/O resources are synchronized by the application program making the requests, except for package Text_Io, which synchronizes requests for output.

9. CAPACITY RESTRICTIONS

The following package specifies the absolute limits on the use of certain language features:

```
with System;
package Limits is

   -- Scanner
  Max_Line_Length                        : constant := 254;

  -- Semantics
  Max_Discriminants_In_Constraint        : constant := 256;
  Max_Associations_In_Record_Aggregate   : constant := 256;
  Max_Fields_In_Record_Aggregate         : constant := 256;
  Max_Formals_In_Generic                 : constant := 256;
  Max_Nested_Contexts                    : constant := 250;
  Max_Units_In_With_Lists                : constant := 256;

  -- Code Generator
  Max_Parameters_In_Call                 : constant := 255;
  Max_Number_Of_Fields_In_A_Record       : constant := 255;
  Max_Number_Of_Entries_In_A_Task        : constant := 255;
  Max_Number_Of_Dimensions_In_An_Array   : constant := 63;
  Max_Nesting_Of_Subprograms_Or_Blocks_In_A_Package_Or_Task
                                         : constant := 14;

  Max_Object_Size                        : constant := (2**32)*System.Bit;

end Limits;
```

10. ATTRIBUTES OF NUMERIC TYPES

This section lists the values returned by attributes that apply to integer types, floating-point types, and the fixed-point type Duration.

10.1. Integer Types

The attributes that apply to integer types—namely, 'First, 'Last, and 'Size—yield the values shown below for the two predefined base types.

Values for type Integer:

Attribute	*Value*
First	-2**31+1
Last	2**31-1
Size	32

Values for type Long_Integer:

Attribute	*Value*
First	-2**63
Last	2**63-1
Size	64

10.2. Floating-Point Types

The attributes that apply to floating-point types yield the following values for the predefined base type Float:

Attribute	*Value*
Digits	15
Emax	204
Epsilon	8.88178419700125E-16
First	-1.79769313486231E+308
Large	2.57110087081438E+61
Last	1.79769313486231E+308
Machine_Emax	1024
Machine_Emin	-1073
Machine_Mantissa	53
Machine_Overflows	True
Machine_Radix	2
Machine_Rounds	False
Mantissa	51
Safe_Emax	1024
Safe_Large	Float'Last - 5.98752092860416E+292
Safe_Small	2.0**(-1025)
Size	64
Small	1.94469227433161E-62

10.3. Type Duration

The attributes that apply to fixed-point types yield the following values for the predefined type Duration:

Attribute	*Value*
Aft	5
Delta	3.0517578125E-05
First	-4.294967296E+09
Fore	11
Large	4.294967296E+09
Last	4.294967296E+09
Mantissa	47
Safe_Large	2.81474976710656E+14
Safe_Small	3.0517578125E-05
Size	48
Small	3.0517578125E-05
Machine_Overflows	True
Machine_Rounds	False